Trabajo Fin de Máster en Programación y Tecnología Software

Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Decision Procedures for the Temporal Verification of Concurrent Data Structures

Alejandro Sánchez

Director: Miguel Palomino Tarjuelo
Colaborador externo: César Sánchez

2010/2011

Calificación obtenida: 10

Autorización de difusión

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el presente Trabajo Fin de Máster: *Decision Procedures for the Temporal Verification of Concurrent Data Structures*, realizado durante el curso académico 2010-2011 bajo la dirección de Miguel Palomino Tarjuelo y con la colaboración externa de dirección de César Sánchez en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

En Madrid, a los 7 días del mes de julio de 2011,

Alejandro Sánchez
Y0403168-S

Miguel Palomino Tarjuelo
Director

Resumen

Los tipos de datos concurrentes básicamente son implementaciones concurrentes de abstracciones de datos clásicas, diseñados específicamente para aprovechar el gran paralelismo disponible en arquitecturas multiprocesador y multinúcleo. La corrección de los tipos de datos concurrentes resulta esencial para demostrar la corrección de sistemas concurrentes en su conjunto. En este trabajo estudiamos el problema de asistir en la automatización de la verificación de propiedades temporales en tipos de datos concurrentes.

La principal dificultad en el razonamiento sobre estos tipos de datos proviene de la interacción entre la alta concurrencia que éstos poseen y la necesidad de manipular al mismo tiempo la memoria dinámica. La mayoría de los enfoques utilizados hasta el momento para la verificación de tipos de datos concurrentes intentan enriquecer *separation logic* para poder hacer frente a la concurrencia, valiéndose del éxito alcanzado por *separation logic* en el razonamiento de algoritmos secuenciales y que manipulan el *heap*.

Este trabajo contiene dos contribuciones. En primer lugar, presentamos un enfoque complementario para la verificación de estructuras de datos concurrentes: partimos de verificación temporal deductiva, una poderosa técnica para razonar sobre sistemas concurrentes, y la enriquecemos para poder lidiar con memoria dinámica. El proceso de verificación utiliza diagramas de verificación y anotación explícita de regiones. Al final, cada prueba se descompone en una secuencia de condiciones de verificación. Los literales que forman parte de cada una de estas condiciones de verificación dependen casi exclusivamente de las estructuras de datos que vayan a ser verificadas.

La segunda, y principal, contribución consiste en dos procedimientos de decisión para tipos de datos específicos: listas concurrentes simplemente enlazadas con cerrojos en cadena y listas concurrentes con saltos. Estos procedimientos de decisión son capaces de razonar sobre regiones, punteros, listas del estilo de *Lisp* o listas ordenadas, permitiendo la verificación automática de las condiciones de verificación generadas previamente.

Aquí demostramos cómo, utilizando nuestra técnica, resulta posible demostrar no solo propiedades de *seguridad*, sino también *viveza* sobre una versión de listas concurrentes, además de la preservación de la forma de listas con saltos por parte de una estructura de datos. Así mismo, el enfoque presentado puede ser fácilmente extendido para razonar sobre un amplio espectro de tipos de datos concurrentes, incluyendo tablas *hash* y grafos.

Palabras claves:

Concurrencia, Verificación Formal, Estructuras de Datos, Procedimientos de Decisión, Diagramas de Verificación

Abstract

Concurrent datatypes are concurrent implementation of classical data abstractions, specifically designed to exploit the great deal of parallelism available in multiprocessor and multicore architectures. The correctness of concurrent datatypes is essential for the overall correctness of the system. In this work we study the problem of aiding in the automation of temporal verification of concurrent datatypes.

The main difficulty to reason about these datatypes comes from the combination of their inherently high concurrency and the manipulation of dynamic memory. Most previous approaches to verification of concurrent datatypes try to enrich separation logic to deal with concurrency, leveraging on the success of separation logic in reasoning about sequential heap algorithms.

This work contains two contributions. First, we present a complementary approach to the verification of concurrent data structures: we start from deductive temporal verification, a very powerful technique to reason about concurrent systems, and enrich it to cope with dynamic memory. The verification process uses verification diagrams and explicit region annotations. In the end, each proof is decomposed into a sequence of verification conditions. The literals involved in these verification conditions depend mainly on the data structured being verified. The second, and main, contribution consists in two decision procedures for specific data-types: concurrent lock-coupling singly-linked lists and concurrent skiplists. These decision procedures are capable of reasoning about regions, pointers, lisp-like lists and ordered lists allowing automatic verification of generated verification conditions.

We show how using our technique we are able to prove not only safety but also liveness properties of a version of concurrent lists and express the preservation of skiplist shape by a data structure. Moreover, the approach we present can be easily extended for using it in the verification of a wide range of similar concurrent datatypes including hash maps and graphs.

Keywords:

Concurrency, Formal Verification, Data Structures, Decision Procedures, Verification Diagrams

Acknowledgments

First of all, I would like to thank César Sanchez for being my advisor, my scientific guide, my inspiration source and the person who drove me into the research world. I am sure that nothing of this work could be possible without his help.

I would also like to thank Miguel Palomino for being the director of this work and for providing me with very useful hints on how to improve this presentation.

Special thanks to my family, who always supported me. Thanks to my parents and my sister who are always with me and, in special, to my grandmother Elena, who taught me my very first letters and has been my early mentor to walk this path of knowledge and science.

Thanks to IMDEA Software for providing me the tools, space and commodities to carry out the current work. In special to all people working there, for their friendship and support.

Finally, I would like to express my special gratitude towards Carolina, Javier and Julian, whose unexpected combined travel abroad gave me the necessary time to finish this work. ت

Thank you all,

Ale.

Contents

1

Introduction

Concurrent data structures [HS08] are an efficient approach to exploit the parallelism of multiprocessor architectures. In contrast with sequential implementations, concurrent datatypes allow the simultaneous access of many threads to the memory representing the data value of the concurrent datatype. Concurrent data structures are hard to design, challenging to implement correctly and even more difficult to formally prove correct.

The main difficulty in reasoning about concurrent datatypes comes from the interaction of concurrency and heap manipulation. The most popular technique to reason about the structures in the heap is separation logic [Rey02]. Leveraging on this success, some researchers [VHHS06, HAN08] have extended this logic to deal with concurrent programs. However, in separation logic disjoint regions are implicitly declared (hidden in the separation conjunction), which makes the reasoning about unstructured concurrency more cumbersome.

Explicit regions allow the use of a classical first-order assertion language to reason about heaps, including mutation and disjointness of memory regions. Regions correspond to finite sets of object references. Unlike separation logic, the theory of sets [WPK09] can be easily combined with other classical theories to build more powerful decision procedures. Classical theories are also amenable of integration into SMT solvers [BSST08]. Moreover, being a classical logic one can use classical Assume-Guarantee reasoning, for example McMillan proof rules [McM99], for reasoning compositionally about liveness properties. In practice, using explicit regions requires the annotation and manipulation of ghost variables of type *region*, but adding these annotations is usually straightforward.

Most of the work in formal verification of pointer programs follows program logics in the Hoare tradition, either using separation logic or with specialized logics to deal with the heap and pointer structures [LQ08, YRS+06, BDES09]. However, extending these logics to deal with concurrent programs is hard, and though some success has been accomplished it is still an open area of research, particularly for liveness.

We propose a complementary approach. We start from temporal deductive verification in the style of Manna-Pnueli [MP95], in particular using general verification diagrams [BMS95, Sip99] to deal with concurrency. This style of reasoning allows a clean separation in a proof between the temporal part (why the interleavings of actions that a set of threads can perform satisfy a certain property) with the underlying data being manipulated. Then, inspired by regional logic [BNR08], we enrich the state predicate language to reason about the different regions in the heap that a program manipulates. Finally, we build decision procedures capable of checking all generated verification conditions generated during our proofs, to aid in the automation of the verification process.

Most previous approaches to verifying concurrent datatypes are restricted to safety properties. In comparison, the method we propose can be used to prove *all liveness* properties, relying on the completeness of verification diagrams.

Verification diagrams can be understood as an intuitive way to abstract the specific aspect of a program which illustrates why the program satisfies a given temporal property. Besides, verification diagrams have been proved to be complete, in the sense that given a program and a temporal formula satisfied by the program then a verification diagram does in fact exist, and sound in the sense that a diagram connecting a program with a formula is in fact a proof that such temporal property is held by the system. We propose the following verification process to show that a datatype satisfies a property expressed in linear temporal logic. First, we build the *most general client* of the datatype, parametrized by the number of threads. Then, we annotate the client and datatype with ghost fields and ghost code to support the reasoning, if necessary. Second, we build a verification diagram that serves as a witness of the proof that all possible parallel executions of the program satisfy the given temporal formula.

The proof is checked in two phases. First, we check that all executions abstracted by the diagram satisfy the property, which can be solved through a fully-automatic finite state model checking method. Second, we must check that the diagram does in fact abstract the program. A verification diagram decomposes a formal proof into a finite collection of verification conditions (VC), each of which corresponds to the effect that a small step in the program has in the data. Then, the verification reduces to verifying this collection of verification conditions. Each concurrent datatype maintains in memory a collection of nodes and pointers with a particular layout. Based on this fact, we propose to use an assertion language whose terms include predicates in specific theories for each layout. To automatize the process of checking the proof represented by a verification diagram it is necessary to use decision procedures for the kind of data structures manipulated. For instance, in the case of singly linked lists, we use a decision procedure capable of reasoning about ideal lists as well as pointers representing lists in memory.

The construction of a verification diagram is a manual task, but it often follows the programmer's intuitive explanation of why the property holds. The activity that we want to automate is checking that the diagram indeed proves the property. Following this idea , in this work we study the automatic verification of VCs for the case of lists and skiplists extending the theory of linked lists presented in [RZ06a]. We present two theories for concurrent singly-linked lists and concurrent skiplists. Besides, we describe a decision procedure for each of them, applying a many-sorted variant of Nelson-Oppen [NO79] combination method.

Logics like those in [LQ08, YRS+06, BDES09] are very powerful to describe pointer structures, but they require the use of quantifiers to reach their expressive power. Hence, these logics preclude a combination a-la Nelson-Oppen or BAPA [KNR05] with other aspects of the program state.

Instead, our solution starts from a quantifier-free theory of single-linked lists [RZ06a], and extends it in a non trivial way with locks, order and sublists of ordered lists. The logics obtained can express list and skiplist-like properties without using quantifiers, allowing the combination with other theories. Proofs for an unbounded number of threads are achieved by parameterizing verification diagrams, splitting cases for interesting threads and producing a single verification condition to generalize the remaining cases. However, in this work we mainly focus on the decision procedure.

The decision procedures that we present here support the manipulation of explicit regions, as in regional logic [BNR08] equipped with *masked regions*, which enables reasoning about disjoint portions of the same memory cell. In this aspect, we use masked regions to "separate"different levels of the same skiplist node.

As we said, we present two theories: $\mathsf{TLL3}$ for concurrent singly-linked lists and TSL_K for concurrent skiplists of height at most K. To illustrate the use of these theories, we sketch the proof of termination of every invocation of an implementation of a lock-coupling concurrent list and the preservation of the skiplist shape of a data structure. We also prove the decidability of $\mathsf{TLL3}$ and TSL_K by showing that they enjoy the finite model property. Moreover, we propose an even more efficient decision procedure for them.

The rest of this work is structured as follows. We conclude Chapter 1 introducing the basic notions of concurrent data structures and decision procedures as well as a description of the progress achieved in these fields. Chapter 2 introduces some basic notions about regional logic and verification diagrams, the building blocks of our verification method. Chapter 3 describes the concurrent data structures for which we have developed decision procedures: concurrent lock-coupling lists and concurrent skiplists. In Chapter 4 we explain the decision procedure for concurrent lock-coupling singly-linked lists. Chapter 5

details the decision procedure for concurrent skiplist. Finally, Chapter 6 gives the conclusions obtained from this work.

1.1 Designing Concurrent Data Structures

Multiprocessor shared-memory systems are capable of running multiple threads, where all threads communicate and synchronize through data structures that reside in shared memory. As one may imagine, concurrent data structures are much more difficult to design than sequential ones due to the way in which involved threads may interleave. In addition, each possible interleave may lead to a different, and possibly unexpected, behavior. In this section we elaborate on some of the aspects described in [MS07] on concurrent systems. We provide an overview of the challenges involved in the design of concurrent data structures as well as a brief description of the most common concurrent datatypes present in the literature.

One of the main sources of difficulty when designing this kind of data structures comes from concurrency. As threads may run concurrently on different processors, they are subject to operating system scheduling policies, page fault, interruptions, etc. Hence, we should reason about the computation as a completely asynchronous process, where steps of different threads can be arbitrarily interleaved.

As an example, imagine we want to implement a *shared counter*. A *shared counter* consists of a shared variable that is concurrently incremented by many threads. A sequential version of such algorithm is depicted in Fig. 1.1(a). This version just fetches the value from counter c and increments it by one, returning the previous value. However, if many threads run this implementation concurrently, then the result may not be the expected one. For instance, consider the scenario at which c is 0 and two threads execute this implementation concurrently. Then, it exists the possibility that both threads read the value 0 from the memory and thus both return 0 as results, which is clearly wrong.

To avoid this problem, a common solution is the use of mutual exclusion locks (usually simply called *mutex* or *lock*). A lock is a construction with the property that, at any given time, nobody owns it or it is owned by a single thread. If a thread T_1 wants to acquire a lock owned by thread T_2, then it must wait until T_2 releases it.

Now, using locks, a correct implementation of the *shared counter* program can be obtained, as shown in Fig. 1.1(b). This new version uses a lock to allow access to the critical section of a single thread at a time. We refer to a program section as *critical* when it provides access to a shared data structure which must not be concurrently executed by more that one thread simultaneously. In the example, the lock clearly prevents two threads to access the critical section. While this is a correct solution, it lacks of good performance. It is easy to achieve a correct implementation just by surrounding the whole code with a lock. However, the performance of such implementation may not differ too much from a sequential one, losing all power provided by concurrency. In the following, we describe some aspects to bear in mind in order to improve performance while preserving correctness, as well as some techniques for verifying the correctness of such data structures.

Procedure SEQ-COUNTER

1: $oldValue := c$
2: $c := oldValue + 1$
3: **return** $oldValue$

End Procedure

(a) Sequential

Procedure CONC-COUNTER

1: **acquire**$(lock)$
2: $oldValue := c$
3: $c := oldValue + 1$
4: **release**$(lock)$
5: **return** $oldValue$

End Procedure

(b) Concurrent

Figure 1.1: Possible implementation of a *shared variable*

1.1.1 Performance Concerns

The *speedup* of a program when run on P different processors is the relation between its execution time on a single processor and its execution time on P processors, concurrently. It can be considered as a measure on how efficient a program uses the processor on which it is running. Ideally, we would like to have linear speedup, however this is not always possible. Data structures whose speedup grows with P are called *scalable*. When designing concurrent data structures, scalability must be taken into account. Naive implementations, as Conc-Counter will surely undermine scalability.

Bearing in mind the example of a shared counter, it is evident that the lock introduces a sequential bottleneck: at every instant, a single thread in allowed to perform useful work. Hence, reducing the size of code sections being sequentially executed is crucial in order to achieve good performance. In the context of locking, we are interested in:

- reducing the number of acquired locks, and
- reducing the lock granularity.

Lock *granularity* measures the number of instructions executed while holding a lock. The fewer instructions, the finer lock granularity. Implementations like the shared counter's one showed above represent an example of a coarse-grain solution.

Another aspect to bear in mind is *memory contention*. Memory contention refers to the overhead in traffic as a result of multiple threads concurrently trying to access the same memory location. For instance, if the lock protecting a critical section is implemented in a single memory location, then a thread trying to access that section must continuously try to access the memory portion where the lock is stored. This problem may be solved if we consider *cache-coherent* multiprocessors. However, using this technique may lead to long waiting times each time a location needs to be modified. As a result, the exclusive ownership of the cache line containing the lock must be repeatedly transfered from one processor to the other.

A final problem with lock-based solutions is that, if a thread holding a lock delays in releasing the lock then all other threads with the intention to acquire the same lock are also delayed. This phenomenon is known as *blocking* and it is quite common on systems with many threads per processor. A possible solution is provided by *non-blocking* algorithms. Non-blocking programs do not use locks and thus the delay of a thread does not imply the delay of other threads. In the following section we describe the main characteristics of lock-based and non-blocking systems.

1.1.2 Synchronization Techniques

Conc-Counter used a lock to prevent many threads accessing the critical section simultaneously. However, sometimes not using locks at all is the best solution for getting concurrency on a system. Here, we describe the two major techniques for accomplishing mutual exclusion on modern concurrent systems: blocking techniques and non-blocking techniques.

Blocking Techniques

As we said, in many cases memory contention and sequential bottleneck can be reduced by reducing the granularity of locking schemes. In fine-grained locking approaches, multiple locks of small granularity are used to protect the critical sections modifying the data structure. The idea is to maximize the amount of time threads are allowed to run in parallel, as far as they do not require to access the same sections of the data structure. This approach can be also used to reduce the contention for individual memory locations. In some cases, this sound natural, as in the case of hash maps. In hash maps, values are hashed to different buckets, one independent of the other, reducing the number of possible conflicts. Hence, placing individual locks on each bucket sounds like a good approach to reduce granularity.

On the other hand, in cases as the shared counter, it is not quite intuitive how contention and sequential bottleneck can be reduced since, abstractly, all operations modify the same part of the data structure. One approach to deal with contention results in spreading each operation accessing the counter in a separate

time interval from the others. A widely used technique to accomplish this is *backoff* [AC89]. However, even reducing the contention, the lock-based implementation of the shared counter example still lacks parallelism and hence it is not scalable. Therefore, more powerful techniques are required.

A possible solution lies on the use of a method known as *combining trees* [GVW89,GGK+83,HLS95, YTL87]. *Combining trees* employ a binary tree with one leaf for each thread. The root of the tree keeps the current value of the counter and intermediate nodes are used to coordinate the access to the root. The idea is that while a thread climbs up to the root of the tree, it combines its effort with the one performed by the other threads. Hence, every time two threads meet on an internal node, their operations are combined into a single operation. Then, one thread waits in the node until a return value is delivered to it while the other one proceeds to the root carrying the operation obtained from the combination.

In the particular case of our shared counter, a winner thread reaching the tree's root modifies the counter in a single atomic operation, performing the action of all combined operations. Then, it goes down once again, delivering the return value to each thread waiting on an internal node. The return values are distributed in such a way that the final effect is as if all operations were executed one after the other at the moment the counter in the root was modified.

Threads waiting on internal nodes repeatedly read on a memory location, waiting for the return value. This sort of waiting is known as *spinning*. An important consequence in the case of a cache-coherent multiprocessor is that the accessed location resides in the local cache of the processor of the waiting thread. Following this approach, no extra traffic is generated. This waiting technique, known as *local spinning*, is very important for scalable performance [MCS91a].

In order to increase performance, *non-uniform memory access* (NUMA) architectures can be used. In NUMA, processors have a mechanism to access their local portions of shared memory faster than they can access the shared memory locations of other processors. A good organization of the data layout on such architectures has a significant impact on performance.

Some techniques, such as combining trees, have the drawback that the total number of threads involved in the system must be known in advance. Moreover, the required space in order to keep the combining tree is proportional to the total number of processors. This way, despite the fact that the technique provides a good through out when the structure is accessed by a significant number of threads, its best case performance under low loads is quite poor, as a result of the fact that until a thread reaches the root, it must traverse $O(\log P)$ tree nodes. On the other hand, the same operation in the lock-based implementation of Fig. 1.1 completes in constant time.

An extra drawback in the combining tree method is that, if coordination between threads going up and down the tree is done incorrectly, it may lead to deadlocks. A *deadlock* is a situation at which two or more threads are executing tasks such that all of them are blocked in a circular fashion, so none of them can progress. Deadlock avoidance is a crucial aspect to bear in mind when designing correct and efficient blocking concurrent data structures and algorithms. In fact, when designing blocking implementations, the number of locks to be taken is a key factor to have in consideration. Enough locks should be used as to ensure correctness while minimizing blocking, allowing other threads to perform concurrent operations in parallel.

Non-blocking Techniques

Non-blocking implementations are designed in order to solve some of the inconveniences present on blocking implementations, avoiding the use of locks. To formalize the idea of non-blocking algorithms, some non-blocking progress conditions are widely described in the literature. Such conditions are known as *wait-freedom*, *lock-freedom* and *obstruction-freedom*. A wait-free [Her91, Lam74] operation must terminate on its own, after a finite number of steps, no matter the behavior of the remaining operations. A lock-free [Her91] operation guarantees that after a finite number of its own steps, some operation terminates: maybe its own operation, or maybe the operation performed by any other thread. Finally, an obstruction-free [HLM03] operation guarantees to terminate after a finite number of its own steps, assuming no interference with other operations.

There exists an evident relation between all these properties. Wait-freedom is stronger than lock-freedom, while lock-freedom is stronger than obstruction-freedom. Clearly, strong progress conditions

are desired over weaker ones. However, weaker conditions are in general simpler and more efficient to design and verify them correct.

Remember that in non-blocking algorithms, no lock can be used. Then, we need to find a new way to implement the concurrent version of the shared counter. [FLP85] (extended to shared memory by [Her91] and [LAA87]) shows that it is not possible to implement a concurrent version using just *load* and *store* instructions. The problem can be solved using a hardware operation that atomically combines both, a *load* and a *store*. In fact, all modern multiprocessors provide one of such synchronization primitives. The most well known are *compare-and-swap* (CAS) [IBM, Int94, SI94] and *load-linked/store-conditional* (LL/SC) [IBM03, Kan89, Sit92]. Operation CAS is described as an algorithm in Fig. 1.2. It receives as arguments a memory location *L* and two values: *E* and *N*. The operation is performed atomically: when called, if the value stored at location *L* matches *E*, then it replaces it with the new value *N* and returns *true*. On the other hand, if the value at *L* does not match *E*, *false* is returned.

```
   Procedure CAS (L, E, N) : Bool
1: if *L = E then
2:     *L := N
3:     return true
4: else
5:     return false
6: end if
   End Procedure
```

Figure 1.2: Description of a CAS operation

In [Her91] it is shown that instructions such as *CAS* and *LL/SC* are universal. This means that, for any data structure, it exists a wait-free implementation in a system that supports such instructions.

Using the CAS instruction it is now possible to implement a non-blocking version of the shared counter program. We just require to load the value of counter *c* and then use CAS to atomically increment the read value by one. In the case the read value does not coincide with the value read when performing the CAS, the operation returns performing no modification to counter c. Hence, we would need to retry until a successful call to CAS is accomplished. Fig. 1.3 shows a non-blocking implementation of the shared counter using CAS. Because the CAS can only fail due to another succeeding *fetch and increment* operation, the implementation is lock-free. However, it is not wait-free as other's thread *fetch and increment* operation can continuously prevent a CAS to succeed.

```
   Procedure NonBlock-Counter
1: repeat
2:     oldValue := c
3:     newValue := oldValue + 1
4: until CAS(&c, oldValue, newValue)
   End Procedure
```

Figure 1.3: Non-blocking shared counter implementation using CAS

Despite the example shown here being simple, in general, designing non-blocking algorithms is more complex than blocking ones, since a thread can use a lock to prevent other threads from interfering while it performs some action. If locks are not used, then the algorithm must be designed in order to be correct despite the actions performed by the other concurrent threads. Currently, in modern architectures, this requirement leads to the use of complicated and costly techniques.

1.1.3 Verification Techniques

In our case, it is quite easy to see that the lock-based implementation of the shared counter behaves exactly the same as the sequential implementation. However, if we consider a more complicated data structure, as a binary tree for instance, then the verification is significantly more difficult. Because of the inherent

difficulty in the design of concurrent datatypes, it is quite easy to fall into an incorrect implementation. Hence, it becomes imperative to rigorously prove that a particular design correctly implements the desired concurrent datatype.

In general, sequential implementations are easier to verify than concurrent ones. For example, the semantics of a sequential data structure can be specified using a set of states and a transition function that given a state, an operation and a set of arguments, returns a new state in addition to the result of applying the operation with the arguments. In fact, if a valid initial state is given, then it is possible to describe all acceptable sequences of operations over the data structure.

Operations on a sequential data structure are executed one after the other, in order. Then, we simply require that the resulting sequence of operations respects the sequential semantic denoted by the set of states and transitions as described above. On the contrary, in the case of concurrent data structures, operations do not really need to be totally ordered. However, correctness for concurrent implementations usually requires the existence of some total order of operations that respects the sequential semantics.

A common condition is Lamport's *sequential consistency* [Lam79]. This condition requires that the total order preserves the order of the operations run by each thread. However, it has a drawback: a data structure constructed by sequential consistent components may not be sequential consistent at all.

Another widely used concept is *linearizability* [HW90], a variation of the concept of *serializability* [BHG87] used in database transactions. Linearizability requires:

1. that the data structure is sequentially consistent, and
2. that the total ordering, which makes it sequentially consistent, respects the real-time ordering between the operations in the execution.

A way of thinking about linearizability is that it requires the user to identify specific points within the algorithms, called *linearization points*, such that if we order the operations according to their linearization points , the resulting order satisfies the desired sequential semantics of the data structure.

It is quite easy to see that Conc-Counter is linearizable. We just need to define the point after the incrementation on c is performed as the linearization point. In the case of NonBlock-Counter, the argument is similar, except that we must define the linearization point considering the semantics of CAS.

The intuitive appeal and modularity of linearizability makes it a very popular correctness condition. Although, most of the concurrent data structures can be shown to be linearizable, on some situations, better performance and scalability can be achieved by considering a weaker condition: *quiescent consistency*. Quiescent consistency condition [AHS94] eliminates the restriction that the total order of operations needs to respect the real-time order of the executed operations, but it requires that every operation executing after a quiescent state must be ordered after every operation executed before the quiescent state. A state is said quiescent if no operations are in progress.

In general, obtaining a formal proof of the correctness for the implementation of a data structure requires:

- a mathematical method for specifying correctness requirements,
- being able to come up with an accurate model of the data structure implementation, and
- ensuring that the proof of the implementation correctness is complete and accurate.

For instance, most linearizability arguments in the literature treat some of these aspects in an informal way. This makes proofs easier to follow and understand. However, the use of some informal description is prone to introduce some error, miss some cases or make some incorrect inferences. On the other hand, rigorous proofs usually contain a great amount of details regarding trivial properties that makes them difficult to write and tedious to read. Hence, computer assisted methods are desired for the formal verification of concurrent implementations. One approach is based on the use of theorem provers to aid in the verification. Another approach consists in the use of model checking. Model checking tools exhaustively verify all possible executions of an implementation, to ensure that all reachable states meet specified correctness conditions. However, some limitations exists on both approaches. Theorem provers usually require significant human insight, while model checking is generally limited by the number of

states it can consider. Therefore, a verification method involving as few human interaction as possible, while being able to verify systems with a possible infinite number of states is desired.

1.1.4 Synchronization Elements

In this section, we describe some basic mechanisms commonly used to achieve correct concurrent implementations: locks, barriers and transactional synchronization mechanisms. Locks and barriers are traditional low level synchronization mechanisms used to prevent some interleaving to happen. For instance, preventing two different threads to access the same section of code at the same time. On the other hand, transactional synchronization mechanisms are used to hide the complex reasoning required to design concurrent data algorithms, letting programmers to think in a more sequential fashion.

As said, locks are low level mechanisms used to prevent processes access the same region of program concurrently. A key issue to bear in mind when designing locks is what to do when a thread tries to get a lock that is already owned by other thread. A possibility is to let threads keep on trying to get the lock. Locks based on this technique are called *spinlocks*. A simple spinlock may use a primitive such as CAS to atomically change the value of a lock from unowned to owned. However, such spinning may cause heavy contention for the lock. An alternative to avoid contention is the use of exponential backoff. In exponential backoff [AC89] a thread that fails in its attempt of getting a lock waits for some time before a new attempt. With every failed attempt, the waiting time is increased. The idea is that threads will spread themselves out in time, resulting in a reduction of contention and memory traffic.

A disadvantage of exponential backoff is that it may happen that a lock remains unlocked for a long time, since all interested threads have been backed-off too much in time. A possible solution to this problem may consist in the use of a queue of interested threads. Locks based on this approach are known as *queuelocks*. Some implementations of queuelocks based on arrays are introduced in [And89, GT90] and then improved using list-based MCS queue locks [MCS91a] and CLH queuelocks [Cra93, MLH94].

Queuelocks also comes in many flavors. There exists abortable implementations of queuelocks where a thread can give up if it is delayed beyond some time limit [Sco02, SI01] or if they just fall into deadlock. On the other hand, preemptive-safe locks [MS98] ensure that an enqueued preempted thread does not prevent the lock to be taken by another running thread.

In some cases, we would like to have locks letting multiple readers access the concurrent data structure. A reader is a process that only extracts information from the data structure, without modifying it at all. Such locks are known as *reader-writer* locks. There exists many kinds of these locks. For instance, reader-writer queuelock algorithms [MCS91b] use a MCS queuelock, a counter for reads and a special pointer for writes. In [KSUH93] readers remove themselves from the lock's queue, keeping a double-linked list and some special locks on the list's nodes. In this case, when a thread removes itself from the list, it acquires a small lock on its neighbor nodes, and redirects the pointers removing itself from the list.

The reader-writer approach can be also generalized to *group mutual exclusion* or *room synchronization*. Under this approach, operations are divided into groups. Operations within the same group can be performed simultaneously with each other, while operations belonging to different groups cannot be executed concurrently. An application of such technique, for instance, could classify the *push* and *pop* operations over stacks on different groups [BCG01]. Group mutual exclusion is introduced in [Jou02] and implementations for *fetch and increment* counters (as the example shown at the beginning of this chapter) are described in [BCG01, KM99].

Another mechanism are barriers. A barrier stops all threads at a given point, allowing them to proceed only after all threads have achieved such point. Barriers are used when the access to the data structure is layered, preventing layer overlapping of different phases.

A barrier can simply be implemented using a counter to keep track of the number of threads that have achieved the barrier position. The counter is initialized with the total number of threads to wait for. Then, every time a thread reaches the barrier, it decrements the counter. Once the counter has reached zero, it let all threads to proceed. This approach still displays the problem of contention, as many threads may be spinning, waiting for the barrier to let them go through. Therefore, special implementations of barriers exist to attack this problem, making threads spin on different locations [III86, HFM88, SMC92]. An alternative approach consists in implementing barriers using diffusing computation trees [DS80]. In this model each thread owns a node in a binary tree. The idea is as follows. Each thread waits in its node

until all its children have arrived. At this moment, the thread communicates its parent that all threads on its branch have arrived. Once all threads have arrived, the root node releases all threads in the tree by disseminating the release information down the tree.

The main purpose for using locks in concurrent programming is to let threads to modify multiple memory locations atomically in such way that no partial result of the computation can be observed by other threads. In this aspect, transactional memory is a mechanism that lets programmers model sections of the code accessing multiple memory locations as a single atomic operation. The use of transactional mechanisms is inspired on the idea of transactions in databases, despite the fact that the problem in memory management is slightly different to the one existing on databases.

An example of transactional memory mechanism for concurrent data structures is *optimistic concurrency control* [KR81]. This approach uses a global lock which is held for a short time at the end of the transaction. However, as one may imagine, such lock is a cause of sequential bottleneck. Ideally, transaction synchronization should be accomplished without the use of locks. Moreover, transactions accessing disjoint sections of the memory should not synchronize with each other at all. A hardware-based transactional memory mechanism was first proposed in [HM93]. An extension to this idea is *lock elision* [RG01, RG02] where the hardware can automatically translate accesses to critical section into transactions that can be executed in parallel.

Despite the effort, up to this moment no hardware support for transactional memory has been developed. Nevertheless, many software based transactional memory approaches have been proposed [ST97b, HFP02, HLMI03, HF03].

1.1.5 Some Concurrent Data Structures

In this section we introduce the basic idea behind some of the most widely spread concurrent data structures.

Stacks and Queues

Stacks and queues are one of the simplest data structures. A concurrent stack is a data structure linearizable with a sequential stack, providing *push* and *pop* operations, respecting the usual LIFO semantics. As one may imagine, many implementations of such data structure exist in the literature. In [MS98] several lock-based linearizable concurrent stacks are presented. These implementations are based on sequential linked lists with a top pointer and a single global lock. As we saw before, structures with a single global lock scale poorly. Besides, if one succeeds in reducing contention on the lock, a sequential bottleneck still exists at the top of the stack.

A first non-blocking implementation of concurrent stacks was presented by Treiber in [Tre86]. In such implementation, a stack is a singly-linked list with a top pointer that is atomically modified through a CAS operation. In [MS98] this algorithm is compared to an optimized non-blocking algorithm based on [Her93] and several lock-based implementations (such as an MCS lock [MCS91a]) in low load scenarios. The results show that the original implementation reaches the best overall performance. However, as the pointer to the top element represents a sequential bottleneck, it offers little scalability as concurrency increases [HSY04].

In fact, in [HSY04] it is shown that any stack implementation can be made more scalable using the elimination technique [ST97a]. This technique allows pairs of operations with reverse semantics (such as *push* and *pop* over a stack) to complete without any central coordination. If this approach is applied using a collision array and adaptive backoff on the shared stack, then it can reach a high level of parallelism with little contention. Following this method, it is also possible to obtain a scalable lock-free implementation of a linearizable concurrent stack.

Using non-blocking algorithms has also its drawbacks. A common problem in many CAS-based implementation is the *ABA* problem [PLJ91]. To illustrate the problem, in Fig. 1.4 we provide a naive implementation of a non-blocking stack using CAS. We assume the stack is implemented using a singly-linked list. Each node in the list stores an element and keeps a *next* pointer to the following element in the stack. Pointer *topPtr* points to the top of the stack. Fig. 1.4 describes the implementation of operations *push* and *pop*. *push* receives as argument the address of the node to be inserted into the stack. The figure

```
   Procedure PUSH (newPtr)
1: repeat
2:     nextPtr := topPtr
3:     newPtr.next := nextPtr
4: until CAS(topPtr, nextPtr, newPtr)
   End Procedure
```

```
   Procedure POP ()
1: repeat
2:     retPtr := topPtr
3:     if retPtr = null then
4:         return null
5:     end if
6:     nextPtr := retPtr.next
7: until CAS(topPtr, retPtr, nextPtr)
8: return retPtr
   End Procedure
```

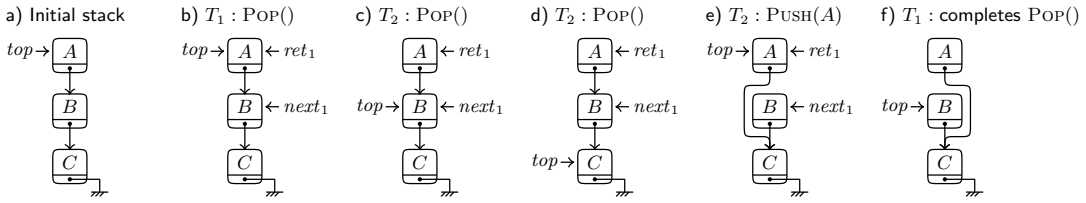


Figure 1.4: Representation of the ABA problem

also describes a scenario at which the ABA problem becomes evident. Fig 1.4 a) shows the initial state of a stack. We assume two threads running concurrently: T_1 and T_2. T_1 wants to perform a *pop*, while T_2 performs two *pop* followed by a *push* of *A*. For simplification, we use *A* to denote both the element and the address of the node storing such element.

Thread T_1 begins executing the *pop* action. However, before it can execute the CAS, T_2 begins to run. Fig. 1.4 b) shows the state of the stack just before T_1 can execute the CAS operation. Fig. 1.4 c), d) and e) depict the situation of the stack just after T_2 has executed *pop*, *pop* and *push*(*A*) respectively. Finally, Fig. 1.4 e) shows the state of the stack once T_1 finishes the original *pop* operation. Notice that according to this implementation, when both thread terminate, the top of the stack contains node *B*, which was eliminated by T_2. This means that we could be accessing an invalid address through the *topPtr* pointer. However, this kind of problems can be solved. A possible solution to this problem consists in equipping the *topPtr* pointer with a version number that is incremented every time the *topPtr* pointer is modified.

As happened with stacks, a concurrent queue is a data structure that is linearizable to a sequential queue and respects the FIFO semantics. [MS98] presents an improved implementation of the single lock implementation of a queue by providing both, the head and tail pointer, with an independent lock. This modification allows the execution of an *enqueue* and a *dequeue* operation in parallel, provided both pointers are different one from each other.

It is possible to implement an array-based lock-free queue for just two processes, one performing only *enqueue* and the other one carrying out just *dequeues* operations, using only load and store operations [Lam83]. A linked-list version of this algorithm is presented in [HS02] and a lock-free array-based implementation assuming unbounded array size is described in [HW90]. Nevertheless, in general, it is quite difficult to implement a nonblocking queue with multiple enqueuers and dequeuers.

In [MS98] a lock-free implementation of a queue based on CAS operations capable of parallel access through both ends is presented. For such purpose it keeps a head and tail pointer, in addition to a dummy pointer. The *enqueue* operation, first uses CAS to add the new node to the end of the list and then uses CAS again to modify the tail pointer to point to the new node. If another *enqueue* operation tries to execute between the first and the second CAS, it will notice that the tail pointer has not been modified yet. A helping technique [Her93] is used to ensure the tail pointer is always at most one node of distance behind the end of the list. Although this implementation is quite simple and efficient, is has a drawback: operations can access nodes that have already been removed from the list. This implies that nodes cannot be freed. Instead, removed nodes are kept in a free-list for reusing them when a new enqueue operation in executed. Nevertheless, in [HLM02, Mic02b] non-blocking memory management techniques are presented in order to overcome this disadvantage.

A generalization of stacks and queues are deques. A deque [Knu68] is a double-ended queue that allows *push* and *pop* at both ends. Lock-based implementations of deques can be easily obtained following the two-lock approach for queues. However it is not easy to come up with a lock-free implementation of deques as shown in [MMS02], even making use of the two-word synchronization primitive *double-compare-and-swap*(DCAS) [Mot86]. One of the few non-blocking implementation of a deque supporting noninterfering operations on both ends is an obstruction-free CAS-based implementation given at [HLM03].

Much of the difficulty in implementing stacks and queues comes from the restrictions on when an inserted element can be removed, depending on the internal order of elements in the data structure. A concurrent pool [Man84] is a data structure which provides *insert* and *delete* operations and where the *delete* operation removes any occurrence of the given element from the structure. An efficient pool can be constructed using quiescently consistent counter implementations [AHS94, SZ96]. In such implementation, elements are kept in an array and a *fetch and increment* operation is used in order to find out the position where an element must be inserted or removed from. Alternatively, a stack-like pool can be also implemented.

Linked Lists

Consider now the implementation of a concurrent data structure supporting *insert*, *remove* and *search* operations. If order does not matter and the operation deals only with a key value, then we have a *multiset* or *set*, depending on whether repeated elements are allowed or not. Moreover, if a data value is associated with every key, then we have a *dictionary* [CLRS01].

In this section we focus on linked-lists used to implement sets. The following sections discuss other concurrent data structures, such as hash tables, search trees and skiplists, that can be used to implement sets.

Considering a single-linked list, the most common approach in order to manage concurrency consists in *lock-coupling* [BS77, Lea99]. In this approach, each node is associated with a different lock. This way, a thread traversing the list releases a node's lock only after it has acquired the lock of the next node in the list, preventing overtaking. Clearly, this approach reduces granularity but it also limits concurrency since insertions and deletions at different locations may delay each other.

A way to evict this problem is to employ ordered lock-free linked lists. The difficulty now resides in ensuring that insertion and deletion keep the validity of the list. The first CAS-based implementation of a list is given in [Val95]. Here, a special node is used in front of every regular node in order to prevent any undesired behavior while manipulating the list. In fact, this lock-free implementation is correct when combined with a memory management [MS95], but this solution is not practical at all. Other approaches use a special bit to mark nodes as deleted (as in [Har01]), but this scheme is only applicable in garbage collectors environments. The problem can be overcome using memory reclamation methods [HLM02, Mic02b] as described in [Mic02a].

Hash Tables

A hash table [CLRS01] can be seen as a resizable array of buckets, where each of them can hold an expected number of elements. Therefore, operations for insertion, deletion and search take constant time. On extensible hash tables, an extra operation for resizing the table is also required. Usually, this latter operation is the most costly of all.

Clearly, it is possible to implement a non-extensible hash table by placing a reader-writer lock on each bucket [Mic02a]. Nevertheless, to keep the good performance, a hash table needs to be extensible [FNPS79].

Already in the eighties, some extensible concurrent hash tables for distributed databases based on two-level locking schemes were introduced [Ell87, HY86, Kum90]. Besides, there exist some highly efficient and extensible hashing algorithms for non-multiprogrammed environments, based on a sequential version of an algorithm for linear hashing [Lit80]. These algorithms are based on a small set of high-level locks, instead of having one lock for each bucket. In addition, it allows concurrent searches while performing the resizing of the hash table. However, concurrent inserts and deletes are not allowed.

Lock-based implementations of concurrent hash tables suffer from the classical drawbacks of lock-based techniques, which becomes even more notorious due to the elaborated resizing methods they require.

Therefore, lock-free implementations of hash tables are desired. Some ideas for reaching a lock-free implementation consist in using an array of buckets, such that each bucket contains a lock-free list. However, this approach does not take into consideration how to deal with extensible arrays, as it is not obvious how to redistribute the elements in a lock-free manner through the buckets of the array. Moving an item from one bucket to another would require two CAS operations to be executed atomically.

Using DCAS it is possible to obtain a lock-free extensible hash table [Gre02]. Nevertheless, this solution has the inconvenient that no all architectures provide the primitive DCAS. In [SS03] an extensible lock-free hash table for concurrent architectures is presented. The idea is to keep a single lock-free list to store the elements instead of a list for each bucket. To preserve the fast access to the elements, the hash table keeps a resizable array of pointers to the list. Operations can then use these pointers to get fast access to internal sections of the list. To keep a constant number of steps per operation, new pointers must be added as the number of elements in the table grows.

Search Trees

As with every concurrent data structure, a concurrent implementation of a tree can be achieved just by placing a single lock protecting the whole structure. This approach can be improved by using a reader-writer lock, such that multiple threads are allowed to search information in the tree, while a single thread at the time is authorized to modify the tree. As expected, the exclusive lock for update operations creates a sequential bottleneck. Concurrency on lock-based implementations can be improved using some fine-locking strategy like placing a lock on each node.

[KL80] presents a concurrent implementation of binary search trees where update operations hold a constant number of locks. Besides, these locks have the property of excluding only other update operations, not blocking search operations. The drawback of this implementation is that it does not attempt to keep the tree balanced.

An alternative approach to allow concurrent modification of a tree based on fine-grained locking consists in forcing an operation to acquire a single lock protecting the subtree that is about to modify. This way, update operations working on disjoint sections of the tree can run in parallel.

For trees, in general, the maintenance operations such as splitting and merging nodes are performed as part of the update operations while inserting or removing elements from the tree. This tight relation between regular operations (*insert* or *delete*) with the maintenance operations is required to preserve the balancing properties of the tree. However, if we relax the balancing property, then maintenance operations can be done separate from regular operations. For instance, B^{link}-trees [Sag86] provide a *compress process* in charge of merging nodes, which can run concurrently with regular operations. By separating maintenance operations from regular ones, we let them run concurrently on different processors or in the background by specialized threads.

The idea of splitting maintenance operations from regular operations was initially described for red-black trees [GS78] and first implemented for AVL trees [AVL62] supporting *insert* and *search* operation in [Kes83]. An implementation supporting also *delete* is provided in [NSSW87]. These implementations improve concurrency by dividing balancing tasks in small transformations that can be applied locally and independently one from each other. It can be shown that the approach presented in [NSSW87] guarantees that each update operation needs at most $O(\log N)$ rebalancing operations for a N-node AVL tree. Similar results have been extended for B-trees [LF95,NSSW87] and red-black trees [BL94,NSS91]. On the non-blocking side we can find implementations based on dynamic software transactional memory mechanisms [Fra03, HLMI03].

Other existing implementations in the literature include studies about failure recovery [ML92] and implementations for *generalized search trees* (GiSTs) [KMH97]. GiSTs simplify the design of search trees without repeating the delicate work involving concurrency. Finally there exists some works [Lar01, Lar02] involving insertion and/or deletion of group of values.

Skiplists

A skiplist [Pug90] is a data structure that implements sets, maintaining several sorted singly-linked lists in memory. In fact, skiplists can be seen as virtual tree structured in multiple levels, where each level consists of a single linked list. The skiplist property establishes that the list at level $i+1$ is a sublist of the list at level i. Each node in a skiplist stores a value and at least the pointer corresponding to the lowest level list. Some nodes also contain pointers at higher levels, pointing to the next element present at that level. The advantage of skiplists is that they are simpler and more efficient to implement than search trees, and search is still (probabilistically) logarithmic.

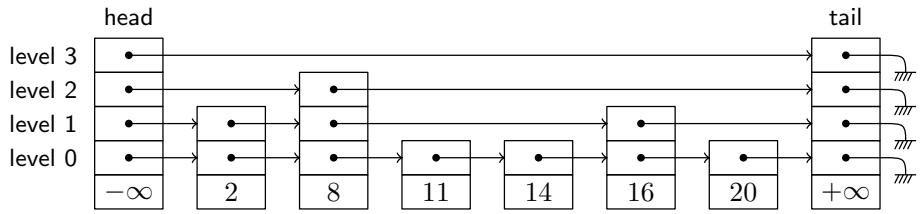


Figure 1.5: A skiplist with 4 levels

Consider for instance the skiplist shown in Fig. 1.5. Contrary to single-linked lists implementations, higher-level pointers allow to *skip* many elements during the search. A search is performed from left to right in a top down fashion, progressing as much as possible in a level before descending. For instance, in Fig. 1.5 a search for value 16 starts at level 3 of node *head*. From *head*, considering level 3, we can reach *tail* in a single jump, which is beyond the possible position of 16. Therefore, the search algorithm moves down one level, to level 2 of node *head*. The *head* successor at level 2 contains value 8, which is smaller than 16, so it is safe to go to node 8 considering level 2. This way, the search continues at level 2 until a node containing a greater value is found. At that moment, the search moves down one level again and so on. The expected logarithmic search follows because the probability of any given node occurring at a certain level decreases by $1/2$ as a level increases (see [Pug90] for an analysis of the running time of skiplists).

Priority Queues

A concurrent priority queue is a data structure linearizable to a sequential priority queue, providing operations *insert* and *delete-min*. Many concurrent implementations of priority queues are based on heaps. When using heaps, insertion is done in bottom-up fashion while deletion of the minimum element is carried out from the root down to the leaves. As one operation proceeds from the leaves and the other from the root, there exists a high probability of achieving deadlock. A possibility to overcome this problem is the use of specialized threads in charge of cleaning up the data structure [BB87]. An alternative consists in implementing *insert* and *delete-min* operations both in a top down fashion [RK88]. An improvement of this algorithm consists in performing consecutive insertions on opposite sides of the heap [Aya90].

In [HMPS96] a heap-based algorithm is presented which does not require the acquisition of multiple locks while traversing the heap. For such purpose, it proceeds by locking a variable that stores the size of the heap, plus a lock over the first and last element in the heap. Other existing implementations [HW91] consist in concurrent priority queues based on concurrent versions of *fibonacci heaps* [FT87].

Non-blocking linearizable heap-based priority queues have been proposed by several authors [Her93, Bar94, IR93]. Even skiplists have been considered for implementing concurrent lock-free versions priority queues [ST03].

Other implementations include priority pools [HW91], where *delete-min* operation is not guaranteed to be linearizable. [Joh91] shows the implementation of a *delete bin* mechanism in charge of keeping elements deleted through *delete-min*, and thus, reducing the load when performing concurrent *delete-min* operations. In [SZ99] priority pools are implemented using skiplist. In general, the relaxed priority pools semantics allows for a significant increment of concurrency.

1.2 Decision Procedures for Pointer Based Data Structures

Pointer based data structured are currently widely used in computer science. In the previous section we have described some of them, including stacks, queues, lists and trees. Here we focus on decision procedures for the automatic verification of properties on these data structures. Many approaches have been suggested and studied in order to reason about them [BRS99, DN03, BPZ05, BR06, Nel83, LQ06, MN05, JJKS97] since the pioneer work of Burstall [Bur72]. However, most of them suffer from extreme difficulties to incorporate reasoning in a wealth of decidable theories over data and pointer values.

The key aspect in many of these approaches resides in the availability of decision procedures to reason about cells (the basic building blocks of the data structure), memories and a reachability notion induced by following pointers. As reachability is not a first-order concept, some features must be added in order to cope with it. Hence, despite the existence of precise and automatic techniques to reason about pointer reachability [JJKS97], not many approaches focus on the combination of such techniques with decision procedures capable of reasoning about data, pointers or locks. As a consequence, approximate solutions have come out and little is known about the combination of such logics with decidable first-order theories to reason about data and pointer values. In some cases, the information over data and pointers is abstracted away to make the use of reachability tools [DN03] possible. In other cases, a first-order approximation of reachability is used [Nel83] so that decision procedures for the theories of pointers and data can be used. In any case, both approaches mentioned above suffer from lack of precision when used.

In [BRS99], the decidability of a logic for pointer-based data structure is proved by showing that it enjoys from finite model property. That is, given a formula, a finite model can be constructed and used to verify the satisfiability of such formula. However, this logic can only reason about reachability on lists, trees and graphs, but it cannot reason about the data stored in the data structure.

A generalization of [BRS99] is presented in [YRS+06], but the emphasis once more is put on expressing complex shape constraints rather than considering the data stored in the data type. Regarding separation logic [Rey02] a decision procedure for lists based on finite model property is described in [BCO04], but it abstracts away the theories over data and pointers. Meanwhile, in combination with predicate abstraction, [BPZ05, BR06] describe decision procedures for logics abstracting away theories over data.

Other logics have been proposed for reasoning about complex data structures [BDES09], but they usually require the use of quantifiers, preventing them from been combined with other theories.

In [RZ06b] TLL is introduced. TLL is a logic for describing single-linked lists as well as a combination based decision procedure capable of reason about reachability. Moreover, it can be extended with available decision procedures for fragments of first-order logic. In TLL it is possible to reason about cells, memory and reachability. In addition, it has the property to be easily combined with other quantifier-free theories capable of reasoning, for instance, about data. In fact, the use of combination schemas [RRZ05] based on Nelson-Oppen [NO79] let them combine TLL with a wide range of available decision procedures for various decidable theories in first-order logic, such as BAPA [KNR05], for example.

The logics we present in this work are an extension of TLL, letting them reason about concurrent lock-based single-linked lists and concurrent skiplists. A key aspect is that both theories remain quantifier-free and with the same combination properties as TLL. Hence, they become perfect candidates for being combined with other theories in order to verify complex systems.

2

Preliminaries

As we said, our final goal is the verification of concurrent data structures. For such purpose, we use verification diagrams. Verification diagrams can be seen as a formula automaton with extra components. Then, the verification process consists in the following steps. First, we construct what we call the *most general client* of the datatype. That is, a program that non-deterministically performs calls to the operations provided by the data structure. Usually, those operations are INSERT, SEARCH and REMOVE. In addition, some non-executable ghost is added to aid the verification process. We can then proceed with the construction of a fair transition system representing the concurrent execution of *M* different instances of such *most general client*. The temporal property we want to verify is expressed as a formula in linear temporal logic and the proof that the system does in fact satisfy the property is represented by the verification diagram. Fig. 2.1 represents the main idea behind our method as well as the role played by decision procedures within this scheme. In this work, we just focus on the development of decision procedures for specific concurrent data structures.

The construction of a verification diagram is not an automatic task. Someone must sketch it, following the reasoning of the system under analysis. An advantage is that, after it has been constructed, verification can be done automatically. The verification that the diagram satisfies the temporal formula can be done

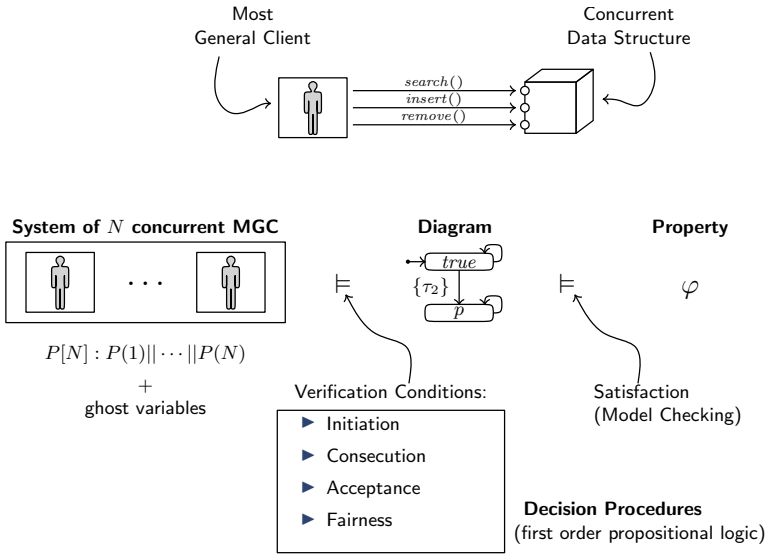


Figure 2.1: Sketch of our proposed approach

just by model checking. On the other hand, to verify that the diagram abstracts the system represented by the fair transition system, a number of verification conditions mush be checked. If we count with a decision procedure capable of reasoning about the theories involved in such verification conditions, then this process can be done automatically. In Chapters 4 and 5 we focus on decision procedures for two specific data types: concurrent lists and skiplists. Meanwhile, in this chapter, we settle the basis of the verification process describing the use of regional logic (distinguishing our approach from the ones based on separation logic) and introducing the concept of verification diagrams, which will later be used in some motivating examples.

2.1 Regional Logic

We are interested in the formal verification of implementations of data structures, in particular in temporal verification (liveness and safety properties) of sequential and concurrent implementations. This verification activity requires to deal with unbounded mutable data. One popular approach to verification of heap programs is separation logic [Rey02].

Separation logic has been widely used up to now in the verification of a wide range of sequential programs. This power of expressiveness is not limited to sequential programs, as some advances on verification of concurrent programs show [GCPV09]. However, some data structures such as skiplists are problematic for separation-like approaches due to the aliasing and memory sharing between nodes at different levels. Based on the success of separation logic some researchers have extended this logic to deal with concurrent programs [Vaf07, HAN08], but concurrent datatypes follow a programming style in which the activities of concurrent threads are not structured according to critical regions with memory footprints. In these approaches based on separation logic memory regions are implicitly declared (hidden in the separation conjunction), which makes the reasoning about unstructured concurrency more cumbersome. Besides, despite the fact that some fragments of separation logic have been proved decidable [BCO04], it is still not completely clear how to integrate separation logic to current SMT solvers.

We use explicit regions to represent the manipulation of memory during the execution of the system. This reasoning is handled by extending the program code with ghost variables of type **rgn**, and ghost updates of these variables. Variables of type **rgn** represent finite sets of object references stored in the heap. Regional logic [BNR08] provides a rich set of language constructs and assertions. However, it is enough for our purposes to use only a small fragment of regional logic. The term **emp** denotes the empty region and $\langle x \rangle$ represents the singleton region whose only object is the one referenced by x. Traditional set-like operators such as $\cup$, $\cap$ and $\setminus$ are also provided and can be applied to **rgn** variables. The assertion language allows reasoning involving mutation and separation. Given two **rgn** expressions r_1 and r_2 we can assert whether they are equal ($r_1 = r_2$), one is contained into the other ($r_1 \subseteq r_2$) or they are completely disjoint ($r_1 \# r_2$).

2.2 Verification Diagrams

We now sketch the important notions from [BMS95, Sip99]. As we said, verification diagrams provide an intuitive way to abstract temporal proofs over fair transition systems (FTS). A FTS Φ is a tuple $\langle V, \Theta, \mathcal{T}, \mathcal{J} \rangle$ where V is a finite set of variables, Θ is an initial assertion, $\mathcal{T}$ is a finite set of transitions and $\mathcal{J} \subseteq \mathcal{T}$ contains the fair transitions. Here we do not consider strong fairness. A *state* is an interpretation of V. We use $\mathcal{S}$ to denote the set of all possible states. A transition $\tau \in \mathcal{T}$ is a function $\tau : \mathcal{S} \to 2^{\mathcal{S}}$, which is usually represented by a first-order logic formula $\rho_\tau(s, s')$ describing the relation between the values of the variables in a state s and in a successor state s'. Given a transition τ, the state predicate $En(\tau)$ denotes whether there exists a successor state s' such that $\rho_\tau(s, s')$.

A computation of Φ is an infinite sequence of states such that:

(a) the first state satisfies Θ;

(b) any two consecutive states satisfy ρ_τ for some $\tau \in \mathcal{T}$;

(c) for each $\tau \in \mathcal{J}$, if τ is continuously enabled after some point, then τ is taken infinitely many times.

We use $\mathcal{L}(\Phi)$ to denote the set of computations of the FTS Φ. Given a formula φ, $\mathcal{L}(\varphi)$ denotes the set of sequences satisfying φ. A FTS Φ satisfies a temporal formula φ if all computations of Φ satisfy φ, i.e., $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\varphi)$.

We use $SF(\varphi)$ to denote the set of all atomic subformulas of φ. A φ-propositional state, denoted by s_φ, assigns a truth-value $s_\varphi[q]$ to each $q \in SF(\varphi)$. Satisfaction of a formula φ over a φ-propositional model $\sigma_\varphi : s_{\varphi,0}, s_{\varphi,1}, \ldots$ is defined inductively where for an atomic subformula q, $s_\varphi \vDash q$ iff $s_\varphi[q] = \mathit{true}$.

A σ-propositional model σ_φ is an infinite sequence of σ-propositional states. The propositional language of φ, $\mathcal{L}^p(\varphi)$, is the set $\{\sigma_\varphi | \sigma_\varphi \vDash \varphi\}$. A propositional projection s^p_φ of a state s is a φ-propositional state such that for each $q \in SF(\varphi)$, $s^p_\varphi \vDash q$ iff $s \vDash q$.

Then, a verification diagram (VD) $\Psi : \langle N, N_0, E, \mu, \eta, \mathcal{F}, \Delta, f \rangle$ is a formula automaton with extra components, abstracting the fair transition system Φ and still preserving the satisfaction of formula φ. All components of a verification diagram are:

- N is a finite set of nodes.
- $N_0 \subseteq N$ is the set of initial nodes.
- $E \subseteq N \times N$ is a set of edges.
- $\mu : N \to F(V)$ is a labeling function mapping nodes to assertions over V.
- $\eta : E \to 2^\tau$ is a labeling function assigning sets of transitions to edges.
- $\mathcal{F} \subseteq 2^{E \times E}$ is an edge acceptance set of the form $\{(P_1, R_1), \ldots, (P_m, R_m)\}$.
- $\Delta \subseteq \{\delta \mid \delta : \mathcal{S} \to \mathcal{D}\}$ is a set of ranking functions from states to a well founded domain $\mathcal{D}$.
- f maps nodes into propositional formulas over atomic subformulas of φ.

If $n \in N$, we use $\mathit{next}(n)$ to denote the set $\{\tilde{n} \in N | (n, \tilde{n}) \in E\}$ and $\tau(n)$ for $\{\tilde{n} \in \mathit{next}(n) | \tau \in \eta(n, \tilde{n})\}$. For each $(P_j, R_j) \in \mathcal{F}$ and for each $n \in N$, Δ contains a ranking function $\delta_{j,n}$. An infinite sequence of nodes $\pi = n_0, n_1, \ldots$ is a path if $n_0 \in N_0$ and for each $i > 0$, $(n_i, n_{i+1}) \in E$. A path π is accepted if for each pair $(P_j, R_j) \in \mathcal{F}$ some edge of R_j occurs infinitely often in π or all edges that occur infinitely often in π are also in P_j. An infinite path π is fair when, for any just transition τ, if τ is enabled on all nodes that appear infinitely often in π then τ is taken infinitely often.

Given a sequence of states $\sigma = s_0, s_1, \ldots$ of Φ, a path $\pi = n_0, n_1, \ldots$ is a trail of σ whenever $s_i \vDash \mu(n_i)$ for all $i \geq 0$. An infinite sequence of states σ is a computation of Ψ whenever there exists an accepting trail of σ such that is also fair. $\mathcal{L}(\Psi)$ is the set of computations of Ψ.

A computation $\sigma = s_{\varphi,0}, s_{\varphi,1,\ldots}$ is a φ-propositional model of Ψ if there is a fair and accepting path $n_0, n_1, \ldots$ in Ψ such that for all $i \geq 0$, $s_{\varphi,i} \vDash f(n_i)$. Finally, $\mathcal{L}^p(\Psi)$ denotes the set of all φ-propositional models of Ψ.

A verification diagram shows that $\Phi \vDash \varphi$ via the inclusions $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi)$. The map f is used to check $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi)$. To show $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\Psi)$ it is enough to prove the following verification conditions:

Initiation: there is at least one initial node from N_0 satisfying the initial conditions of the fair transition system Φ.

Consecution: Any τ-successor of a state satisfying $\mu(n)$ does satisfy the labeling predicate of some successor node of n. This means that for every node $n \in N$ and transition $\tau \in \mathcal{T}$,

$$\mu(n)(s) \ \wedge \ \rho_\tau(s, s') \ \rightarrow \ \mu(\mathit{next}(n))(s')$$

Acceptance: For each pair (P_j, R_j) member of the acceptance list and for any $e = (n_1, n_2) \in E$ and $\tau \in \mathcal{T}$, when we take τ from an arbitrary state s, if $e \in P_j \setminus R_j$ then δ_j cannot increase, while if $e \notin P_j \cup R_j$ then function δ_j must decrease. That is:

1. if $(n_1, n_2) \in P_j \setminus R_j$ then

$$\rho_\tau(s, s') \ \wedge \ \mu(n_1)(s) \ \wedge \ \mu(n_2)(s') \ \rightarrow \ \delta_{j,n_1}(s) \succeq \delta_{j,n_2}(s')$$

2. if $(n_1, n_2) \notin P_j \cup R_j$ then

$$\rho_\tau(s, s') \ \wedge \ \mu(n_1)(s) \ \wedge \ \mu(n_2)(s') \ \rightarrow \ \delta_{j,n_1}(s) \succ \delta_{j,n_2}(s')$$

Fairness: For each $e = (n_1, n_2) \in E$ and $\tau \in \eta(e)$:

1. τ is guaranteed to be enabled in every $\mu(n_1)(s)$:

$$\mu(n_1)(s) \ \rightarrow \ En(\tau)$$

2. Any τ-successor of a state satisfying $\mu(n_1)$ satisfies the label of some node in $\tau(n_1)$:

$$\mu(n_1)(s) \ \wedge \ \rho_\tau(s, s') \ \rightarrow \ \mu(\tau(n_1))(s')$$

Satisfaction:

1. For all $n \in N$, if $s \vDash \mu(n)$ then $s^p_\varphi \vDash f(n)$
2. $\mathcal{L}^p(\Psi) \subseteq \mathcal{L}^p(\varphi)$

Some verification conditions such as *satisfaction* can be checked through model checking, just verifying whether the language accepted by the diagram is contained into the language of the temporal formula, or equivalently, checking if the intersection with the language of the negation of the formula is empty.

The remaining verification conditions can be automatically verified if we count with an appropriated decision procedure. In such case, the components involved in the formulas depend on the data structure we are verifying. In the following chapters, we present two concurrent data structures: concurrent lock-coupling lists and concurrent skiplists. Moreover, we present some examples of properties we would like to verify over these datatypes in addition to adequate decision procedures for each of them.

```
   global
     Int tick := 0
     Set⟨Int⟩ announced := ∅ ———————— Set⟨Int⟩ lower := ∅

   procedure MutExc
     Int ticket
   begin
1: loop
2:     nondet
3:     ⟨ ticket := tick++ ⟩ ———————— if k ∉ announced
       ⟨ announced.add(ticket) ⟩          lower.add(ticket)
4:     await(announced.min == ticket)
5:     critical
6:     announced.remove(ticket) ———————— if my_id = k
                                            lower = announced − ticket
                                          else lower = lower − ticket
7: end loop
   end procedure
```

Figure 2.2: MutExc: A mutual exclusion algorithm (left), and annotations (right)

We now illustrate the use of generalized verification diagrams by showing the proof of a response property of a simple mutual exclusion algorithm. The algorithm is structured the following way. Each thread that wants to access the critical section acquires an increasing number (ticket) and announces its intention to enter the critical section by adding its ticket to a shared global set of tickets. Then, each thread waits until its ticket number becomes the lowest value in the set before entering the critical section. After a thread leaves the critical section it removes its ticket from the set. For the sake of simplicity, we assume that all operations are atomic.

Fig. 2.2 (left) shows MutExc, a program that implements this mutual exclusion protocol to protect access to the critical section at line 5, using two global variables. The *Int* variable *tick* stores the shared increasing counter. The *Set* variable *announced* stores the ticket numbers of all threads that are trying to access the critical section. We want to prove that if an arbitrary thread k requests access to the critical section, eventually it succeeds. To aid in the verification, ghost variable *lower* is used to keep the track of all requests that have been done before thread k. We use the cardinality of such set as a ranking function to prove that eventually, the request of thread k is taken into consideration. In general, ghost annotations are introduced in order to keep track of special conditions and values that aid in the verification process. Variable *ticket* stores the ticket number assigned to the thread while variable *pc* keeps the program counter. If S is a set of variables $v_1, \ldots, v_n$, we use $pres(S)$ to denote that the value of all variables in S

$$V: \quad \{tick,\ announced,\ ticket^{[1]},\ ticket^{[2]},\ pc^{[1]},\ pc^{[2]}\}$$

$$\begin{array}{lll}
\rho_{\tau_1^{[1]}}: pc^{[1]} = 1 \wedge pc^{[1]} = 2 \wedge & & pres(V \setminus \{pc^{[1]}\}) \\
\rho_{\tau_2^{[1]}}: pc^{[1]} = 2 \wedge pc^{[1]} = 3 \wedge & & pres(V \setminus \{pc^{[1]}\}) \\
\rho^{\mathsf{T}}_{\tau_3^{[1]}}: pc^{[1]} = 3 \wedge pc^{[1]} = 4 \wedge & k \in announced & \\
& \wedge\ ticket'^{[1]} = tick + 1 & \\
& \wedge\ tick' = tick + 1 & \\
& \wedge\ announced' = announced \cup \{ticket^{[1]}\} & \wedge\ pres(V \setminus \{pc^{[1]}, ticket^{[1]}, announced\}) \\
\rho^{\mathsf{F}}_{\tau_3^{[1]}}: pc^{[1]} = 3 \wedge pc^{[1]} = 4 \wedge & k \notin announced & \\
& \wedge\ ticket'^{[1]} = tick + 1 & \\
& \wedge\ tick' = tick + 1 & \\
& \wedge\ announced' = announced \cup \{ticket^{[1]}\} & \\
& \wedge\ lower' = lower \cup \{ticket^{[1]}\} & \wedge\ pres(V \setminus \{pc^{[1]}, ticket^{[1]}, announced, lower\}) \\
\rho_{\tau_4^{[1]}}: pc^{[1]} = 4 \wedge pc^{[1]} = 5 \wedge & announced.min = ticket^{[1]} & \wedge\ pres(V \setminus pc^{[1]}) \\
\rho_{\tau_5^{[1]}}: pc^{[1]} = 5 \wedge pc^{[1]} = 6 \wedge & & pres(V \setminus \{pc^{[1]}\}) \\
\rho^{\mathsf{T}}_{\tau_6^{[1]}}: pc^{[1]} = 6 \wedge pc^{[1]} = 7 \wedge & my_id = k & \\
& \wedge\ announced' = announced \setminus \{ticket^{[1]}\} & \\
& \wedge\ lower' = announced \setminus \{ticket^{[1]}\} & \wedge\ pres(V \setminus pc^{[1]}, announced, lower) \\
\rho^{\mathsf{F}}_{\tau_6^{[1]}}: pc^{[1]} = 6 \wedge pc^{[1]} = 7 \wedge & my_id \neq k & \\
& \wedge\ announced' = announced \setminus \{ticket^{[1]}\} & \\
& \wedge\ lower' = lower \setminus \{ticket^{[1]}\} & \wedge\ pres(V \setminus pc^{[1]}, announced, lower) \\
\rho_{\tau_7^{[1]}}: pc^{[1]} = 7 \wedge pc^{[1]} = 1 \wedge & & pres(V \setminus \{pc^{[1]}\})
\end{array}$$

$$\Theta: \quad tick = 0 \wedge announced = \emptyset \wedge lower = \emptyset \wedge pc^{[1]} = 1 \wedge pc^{[2]} = 1$$

Figure 2.3: Fair transition system for MutExc

is preserved by the transition. That is, that $v'_1 = v_1 \wedge \cdots \wedge v'_n = v_n$.

For simplicity, we assume a system made just from two threads: T_1 and T_2. We would like to prove that whenever T_1 shows it interest to enter the critical section, then it eventually succeeds.

Throughout this work we use $at\_p_n^{[k]}$ to denote that thread k is at line n of program p. Similarly, we use $at\_p_{n..m}^{[k]}$ to represent $\bigwedge_{i\in[n..m]} at\_p_i^{[k]}$. If v is a local variable of a program, we use $v^{[k]}$ to denote the local instance of variable v owned by thread k. As usual, we use primed variables to describe the values of the variables after the transition is taken.

We can define the fair transition representing an instance of MutExc for two threads as shown in Fig. 2.3. Here we limit ourselves to show just the transition relations for thread T_1. A set of analog transition relations are required for thread T_2.

Before we describe the property and the diagram, we introduce some notation to ease the proof. We use $active(i)$ for $at\_\text{MutExc}_{4,5,6}^{[i]}$, $wants(i)$ for $at\_\text{MutExc}_3^{[i]}$, and $critical(i)$ for $at\_\text{MutExc}_5^{[i]}$. Diagrams for the verification of safety properties are quite easy to construct. For instance, imagine we want to verify mutual exclusion of the system. Then, we require to verify a set of invariants. For instance, below $\varphi_1(i)$ describes the fact that every time a thread i has a valid ticket, the value of this ticket is smaller than $tick$. Formula $\varphi_2(i, j)$ establishes that two different threads cannot have the same ticket. Formula $\varphi_3(i)$ specifies that if a thread is in $active(k)$ then its ticket is in the announced set. Formula $\varphi_4(i)$ establishes that the thread in the critical section owns the lowest ticket. Finally, mutual exclusion is expressed as φ_{mutex}:

$$
\begin{array}{lcl}
\varphi_1(i) & \hat{=} & \Box(active(i) \rightarrow ticket(i) < tick) \\
\varphi_2(i, j) & \hat{=} & \Box(i \neq j \wedge active(i) \wedge active(j) \rightarrow ticket(i) \neq ticket(j)) \\
\varphi_3(i) & \hat{=} & \Box(active(i) \rightarrow ticket(i) \in announced) \\
\varphi_4(i) & \hat{=} & \Box(critical(i) \rightarrow min(announced) = ticket(i)) \\
\varphi_{mutex}(i, j) & \hat{=} & \Box\,(i \neq j \rightarrow \neg(critical(i) \wedge critical(j)))
\end{array}
$$

It is easy to see that these invariants hold. In this work we are not interested in describing parametric systems. Thus, given a system, we assume a fixed number of threads running concurrently. This way, all properties need to be verified for every thread in the system. If we would like to construct a diagram for, let say, $\varphi_1(T_1)$, then it would look like the one shown in Fig. 2.4. Despite verification can be done automatically, the construction of a diagram remains as a manual task. Hence, it is the user the one who must sketch it, following its intuition, trying to capture the behavior of the program.

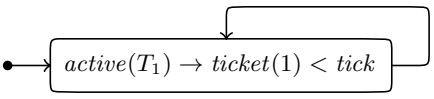


Figure 2.4: Verification diagram for $\varphi_1(T_1)$

We show now how to verify liveness properties. The program MutExc satisfies that every thread that wants to enter the critical section eventually does, formally expressed in LTL using the following response property [MP95]:

$$\psi(k) \hat{=} \Box(wants(k) \rightarrow \Diamond critical(k))$$

In the verification we use the ghost variables and ghost updates, shown in Fig. 2.2 (right). This variables allow to keep track of important aspects of the history of the computation, in this case, the set of lower tickets (with respect to $ticket^{[k]}$).

We show how to construct the diagram for verifying $\psi(T_1)$, that is, that the property holds for thread T_1. Two diagrams are shown in Fig. 2.5. The first one labels nodes with an informal description of the state they represent. The second one is labelled with program positions and is the one used for the

verification. Formally, the diagram is defined by:

$$\begin{aligned} N &\triangleq \{n_i \mid 1 \leq i \leq 6\} \\ N_0 &\triangleq \{n_1\} \\ \mathcal{F} &\triangleq \{(P,R)\} \text{ with } R = \{(n_4, n_1)\} \text{ and } P = E - (R \cup \{(n_6, n_3)\}) \\ f(n) &\triangleq \mu(n) \\ \delta(n,s) &\triangleq |lower| \end{aligned}$$

The edges shown in Fig. 2.5 represent the set E. Similarly, μ an η are defined, respectively, as the functions that label nodes and edges with the formulas and transitions depicted in the same figure.

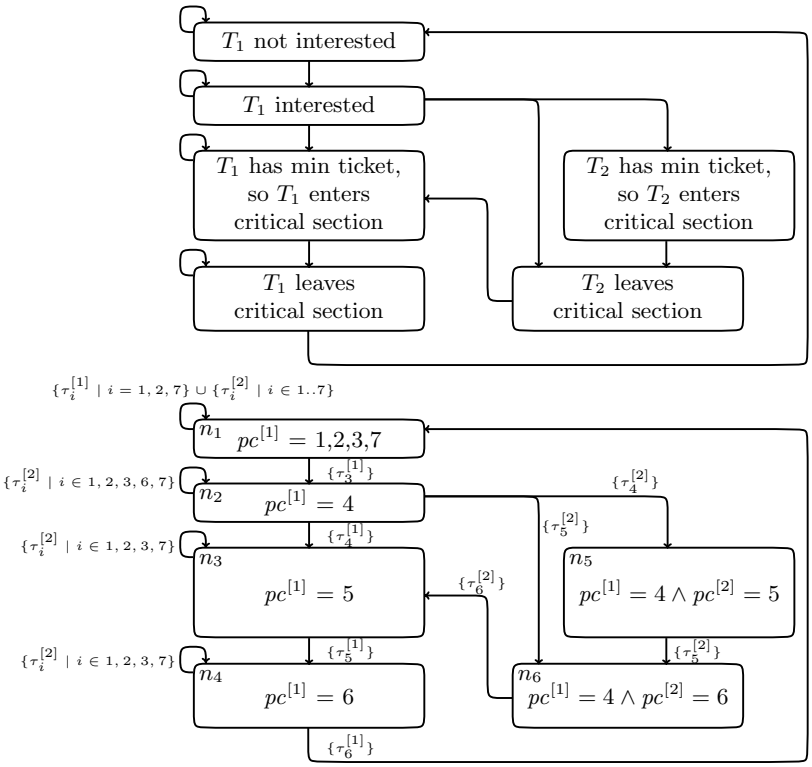


Figure 2.5: Verification diagram for $\psi(T_1)$

3

Concurrent Lists and Skiplists

In this work we focus on decision procedures for two pointer based data structures: concurrent lock-coupling singly-linked lists and concurrent skiplists. In this chapter, we introduce both data types, describe their structure and give the main operations that manipulate them. Besides, for each data type we sketch a proof requiring temporal reasoning. This will let us present the components we will require in our decision procedures for each data type.

3.1 Concurrent Lock-Coupling Lists

Lock-coupling concurrent lists [HS08, VHHS06] are ordered lists with non-repeating elements, in which each node is protected by a lock. A thread advances through the list acquiring the lock of the node it visits. This lock is only released after the lock of the next node has been acquired. The following *List* and *Node* structures are used to maintain the data of a concurrent list:

class *List* {*Node** *list*; }
class *Node* {*Value val*; *Node** *next*; *Lock lock*; }

A *List* contains one field pointing to the *Node* representing the head of the list. A *Node* consists of a value, a pointer to the next *Node* in the list and a lock. We assume that the operating system provides the operations *lock* and *unlock* to acquire and release a lock. Every list has two sentinel nodes, *Head* and *Tail*, with phantom values representing the lowest and highest possible values. For simplicity, we assume such nodes cannot be removed or modified. Fig 3.1 presents a concurrent lock-coupling skiplist.

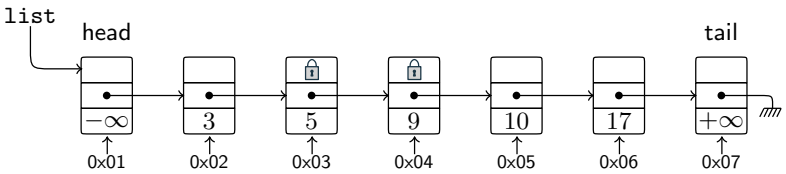


Figure 3.1: A lock-coupling singly-linked list with elements 5 and 9 locked

Concurrent Lock-Coupling Lists are used to implement sets, so they offer three operations:

- LOCATE, shown as Algorithm 3.1, finds an element traversing the list. This operation returns the pair consisting of the desired node and the node that precedes it in the list. If the element is not found the *Tail* node is returned as the current node. A SEARCH operation, shown as Algorithm 3.2, that decides whether an element is in the list can be easily extended from LOCATE.

- INSERT, shown as Algorithm 3.3, inserts a new element in the list, using LOCATE to determine the position at which the element must be inserted. The operation *add* returns *true* upon success, otherwise it returns *false*.
- REMOVE, depicted as Algorithm 3.4, deletes a node from the list by redirecting the next pointer of the previous node appropriately.

Algorithm 3.1 Locate program for concurrent lock-coupling singly-linked lists

```
1:  procedure LOCATE(Value e)
2:      prev := Head
3:      prev.lock()
4:      curr := prev.next
5:      curr.lock()
6:      while curr.val < e do
7:          prev.unlock()
8:          prev := curr
9:          curr := curr.next
10:         curr.lock()
11:     end while
12:     return (prev, curr)
13: end procedure
```

Algorithm 3.2 Search for concurrent lock-coupling singly-linked lists

```
1:  procedure SEARCH(Value e)
2:      prev, curr := LOCATE(e)
3:      if curr.val = e then
4:          result := true
5:      else
6:          result := false
7:      end if
8:      curr.unlock()
9:      prev.unlock()
10:     return result
11: end procedure
```

Algorithm 3.3 Insertion for concurrent lock-coupling singly-linked lists

```
1:  procedure INSERT(Value e)
2:      prev, curr := LOCATE(e)
3:      if curr.val ≠ e then
4:          aux := new Node(e)
5:          aux.next := curr
6:          prev.next := aux
7:          result := true
8:      else
9:          result := false
10:     end if
11:     prev.unlock()
12:     curr.unlock()
13:     return result
14: end procedure
```

Algorithm 3.4 Remove for concurrent lock-coupling singly-linked lists

```
1:  procedure REMOVE(Value e)
2:      prev, curr := LOCATE(e)
3:      if curr.val = e then
4:          aux := curr.next
5:          prev.next := aux
6:          result := true
7:      else
8:          result := false
9:      end if
10:     prev.unlock()
11:     curr.unlock()
12:     return result
13: end procedure
```

Algorithm 3.5 Most general client for concurrent lock-coupling singly-linked lists

```
1:  procedure MGC
2:      while true do
3:          e := NondetPickElem
4:          nondet
                [ call SEARCH(e)
                  or
5:                call INSERT(e)
                  or
                  call REMOVE(e) ]
6:      end while
7:  end procedure
```

We can also construct the most general client of the concurrent-list datatype: the program MGC that repeatedly chooses non-deterministically a method and its parameters. Such client is shown as Algorithm 3.5.

Example 3.1 (Termination of rightmost thread)
Imagine we want to prove that on a lock-coupling concurrent list as the one introduced in this chapter, the thread owning the nearest lock to the end of the list eventually terminates. To prove it, we construct a fair transition system $\mathcal{S}[N]$ parametrized by the total number of threads N, in which all threads run MGC. Let $\psi(k)$ be the temporal formula that describes that if thread k owns the last lock in the list (i.e., the lock nearest to node *Tail*) then it eventually terminates. The verification problem is then casted as $\mathcal{S}[N] \models \psi(k)$, for all N. A sketch of a verification diagram is depicted in Fig. 3.2. We say that a thread is the rightmost owning a lock when there is no other thread owning a lock that protects a *Node* closer to the tail.

Each diagram node is labeled with a predicate. This predicate captures the set of states of the transition system that the node abstracts. Edges represent transitions between states abstracted by the nodes. For the sake of simplicity, in the diagram at Fig. 3.2 we have replaced the predicates labeling each node by an informal description of what they represent.

Node n_1 denotes all states in which thread k has not acquired a lock yet, or it does not own the rightmost lock. Node n_2 represents states in which thread k has acquired the rightmost lock in the list and it is about to execute a non-blocking statement. Node n_3 denotes the states at which thread k is the rightmost thread owning a lock, and no other thread is blocking it. Node n_4 represents the states on which thread k is about to acquire a lock so it could get blocked. Node n_5 denotes the states at which thread k

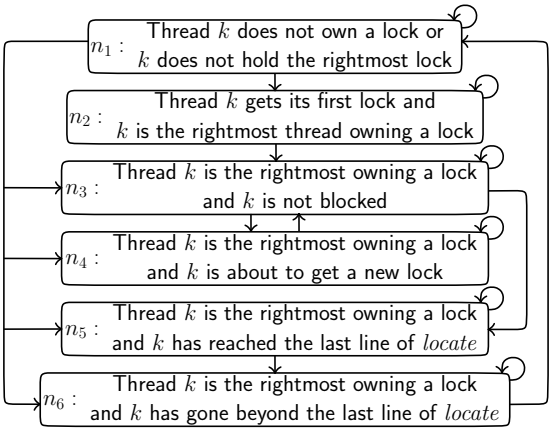


Figure 3.2: Sketched verification diagram for $\mathcal{S}[N] \vDash \psi(k)$

has reached the last line of program Locate meaning that no more locks needs to be acquired. Finally, node n_4 denotes the state at which thread k has gone beyond the last line of program Locate. At this point, the operations done by thread k consists only in modifying some pointers and releasing locks.

Since we assume that all threads are running program MGC, by "thread j terminates" we mean that thread j has reached the last program line of Locate. Notice that once thread j has arrived to the last line of Locate it is easy to show that j cannot be blocked.

Checking the proof represented by the verification diagram requires two activities. First, to show that all traces of the diagram satisfy the temporal formula $\psi(k)$, which can be performed by finite state model checking. Second, to prove that all computations of $\mathcal{S}[N]$ are traces of the verification diagram. This process involves the verification of formulas built from several theories. For instance, considering the execution of line 6 of program Insert we should verify that the following condition holds:

$$at\_\textsc{Insert}_5^{[k]} \wedge \mathit{IsLast}(k) \wedge \begin{pmatrix} r' = r \cup \left\langle aux^{[k]} \right\rangle & \wedge \\ prev'^{[k]}.next = aux^{[k]} \end{pmatrix} \rightarrow at'\_\textsc{Insert}_6^{[k]} \wedge \mathit{IsLast}'(k) \tag{3.1}$$

The predicate $prev'^{[k]}.next = curr^{[k]}$ is in the theory of pointers, while $r' = r \cup \left\langle curr^{[k]} \right\rangle$ is in the theory of regions. Moreover, some predicates belong to a combination of theories, like $\mathit{IsLast}(k)$, which among other things establishes that $\mathit{List}(h, x, r)$ holds. $\mathit{List}(h, x, r)$ expresses that in heap h, starting from pointer x, the pointers form a list of elements following the *next* field, and that all nodes in this list form precisely the region r. ∗

Our intention here is not to show a full example or a complete verification condition, but just to give the reader the intuition of the construction our decision procedure should be able to deal with. At this point, it should be clear that we must deal with elements, addresses, thread identifiers, nodes, memory assignments, regions and locks. To accomplish the automatic verification, we must build a suitable decision procedure involving all these theories. Such decision procedure is described in Chapter 4.

3.2 Concurrent Skiplists

A skiplist [Pug90] is a data structure that implements sets, maintaining several sorted singly-linked lists in memory. Skiplists are structured in multiple levels, where each level consists of a single linked list. The skiplist property establishes that the list at level $i + 1$ is a sublist of the list at level i. Each node in a skiplist stores a value and at least the pointer corresponding to the lowest level list. Some nodes also contain pointers at higher levels, pointing to the next element present at that level. The advantage of skiplists is that they are simpler and more efficient to implement than search trees, and search is still (probabilistically) logarithmic.

In this section we present a simple concurrent implementation of skiplists using lock-coupling [HS08] to acquire and release locks. This implementation can be seen as an extension of concurrent lock-coupling lists [HS08, Vaf07] to multiple layers of pointers. This algorithm imposes a locking discipline, consisting of acquiring locks as the search progresses, and releasing a node's lock only after the lock of the next node in the search process has been acquired. A naïve implementation of this solution would equip each node with a single lock, allowing multiple threads to access simultaneously different nodes in the list, but protecting concurrent accesses to two different fields of the same node. The performance can be improved by carefully allowing multiple threads to simultaneously access the same node at different levels. We study here an implementation of the latter solution in which each node is equipped with a different lock at each level.

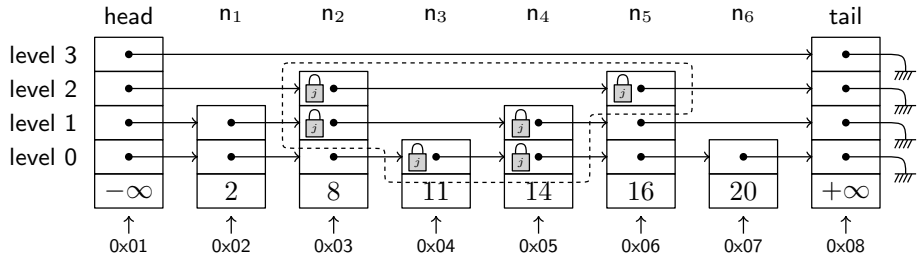


Figure 3.3: A skiplist with the masked region given by the fields locked by thread j

At execution time a thread uses locks to protect the access to only some fields of a given node. A precise reasoning framework needs to capture those portions of the memory protected by a set of locks, which may include only *parts* of a node. Approaches based on strict separation (separation logic [Rey02] or regional logic [BNR08]) do not provide the fine grain needed to reason about individual fields of shared objects. We now introduce the concept of *masked regions* to describe regions and the fields within. A masked region consists of a set of pairs formed by a region (*Node* cell) and a field (a skiplist level):

$$\mathbf{mrgn} \triangleq 2^{Node \times \mathbb{N}}$$

We call the field a mask, since it identifies which part of the object is relevant. For example, in Fig. 3.3 the region within dots represents the area of the memory that thread j is protecting. This portion of the memory is described by the masked region $\{(n_2,2),(n_5,2),(n_2,1),(n_4,1),(n_3,0),(n_4,0)\}$. As with regional logic, an empty set intersection denotes separation. In masked regions two memory nodes at different levels do not overlap. This notion is similar to data-groups [Lei98].

We now present the pseudo-code declaration of the *Node* and *SkipList* classes:

```
class  SkipList  {  Node* head; Node* tail;  }
class  Node      {  Value val;
                    Key key;
                    Array⟨Node*⟩(K) next;
                    Array⟨Node*⟩(K) lock;    }
```

Throughout the present work we use boxes to denote ghost code added for verification purposes. Note that the structure is parametrized by a value K, which determines the maximum possible level of any node in the modeled skiplist. The fields *val* and *key* in the class *Node* contain the value and the key of the element used to order them. Then, we can store key-value pairs, or use the skiplist as a set of arbitrary elements as long as the key can be used to compare. The *next* array stores the pointers to the next nodes at each of the possible K different levels of the skiplist. Finally, the *lock* array keeps the locks, one for each level, protecting the access to the corresponding *next* field.

The *SkipList* class contains two pointer fields: *head* and *tail* plus a ghost variable field *r*. Field *head* points to the first node of the skiplist, and *tail* to the last one. Variable *r*, only used for verification purposes, keeps the (masked) region represented by all nodes in the skiplist with all their levels. In this

Algorithm 3.6 Unfair implementation of concurrent skiplist operation INSERT

```
1:  procedure INSERT(SkipList sl, Value newval)
2:      Vector⟨Node*⟩upd[0..K − 1]                          ——— L := L{me ← sl.head}
3:      lvl := randomLevel(K)                                    U := U{me ← sl.tail}
4:      Node* pred := sl.head                                    H := H{me ← K − 1}
5:      pred.locks[K − 1].lock()
6:      Node* curr := pred.next[K − 1]
7:      curr.locks[K − 1].lock()
8:      for i := K − 1 downto 0 do
9:          if i < K − 1 then
10:             pred.locks[i].lock()                         ——— U := U{me ← pred.next[i + 1]}
11:             if i ≥ lvl then
12:                 curr.locks[i + 1].unlock()               ——— H := H{me ← i}
13:                 pred.locks[i + 1].unlock()
14:             end if
15:             curr := pred.next[i]
16:             curr.locks[i].lock()
17:         end if
18:         while curr.val < newval do
19:             pred.locks[i].unlock()                       ——— if (i ≥ lvl){L := L{me ← curr}}
20:             pred := curr
21:             curr := pred.next[i]
22:             curr.locks[i].lock()
23:         end while
24:         upd[i] := pred
25:     end for
26:     Bool valueWasIn := (curr.val = newval)
27:     if valueWasIn then
28:         for i := 0 to lvl do
29:             upd[i].next[i].locks[i].unlock()
30:             upd[i].locks[i].unlock()
31:         end for
32:     else
33:         x := CreateNode(lvl, newval)
34:         for i := 0 to lvl do
35:             x.next[i] := upd[i].next[i]
36:             upd[i].next[i] := x
37:             x.next[i].locks[i].unlock()
38:             upd[i].locks[i].unlock()
39:         end for
40:     end if                                               ——— L = L{me ← null}
41:     return ¬valueWasIn
42: end procedure
```

implementation, *head* and *tail* are sentinel nodes, with *key* = $-\infty$ and *key* = $+\infty$, respectively. For simplicity, these nodes are not eliminated during the execution and their *val* field remains unchanged.

When considering concurrent datatypes, a craft design must be taken in consideration. Apparently correct implementations may lead to undesired behaviors. For instance, consider the implementation for INSERT given as Algorithm 3.6. This implementation does not properly works under the assumption of

strong fairness, since a thread may prevent other to progress under the assumption of an unfair scheduler.

Example 3.2
We show that the implementation described as Algorithm 3.6 does not ensure termination of all threads under the assumption of weak-fairness. The program has been enriched with ghost arrays L and U from thread identifiers to addresses and an array H from thread identifiers to skiplist levels. These three global arrays are used to store for each thread the addresses and level that delimit the section of the skiplists that can be potentially be modified by a thread. We use $e = x[i]$ to denote that e is the value stored in array x at position i. We also use $y = x\{i \leftarrow e\}$ to denote that y is the array resulting from taking x and modifying position i to store element e. This means that $y[i] = e$ while $y[j] = x[j]$ for any other j different from i. This way, $L[t]$ stores the lowest address than can be modified by thread t. Similarly, $U[t]$ denotes the upper address of the region modifiable by t. Meanwhile, $H[t]$ is the highest level that can be modified by thread t. In any case, we assume any thread can use *me* to refer to its thread identifier.

Notice that skiplists levels are numerated starting from 0. Then, a node at level 0 is said to have height 1. Similarly, a node that goes up to level 1 is said to be of height 2 and so on.

Consider now the skiplist described in Fig. 3.4 and a system with two thread: T_1 and T_2. Imagine T_1 wants to insert a node with value 14 and height 1, while T_2 tries to insert value 16 with height 2.

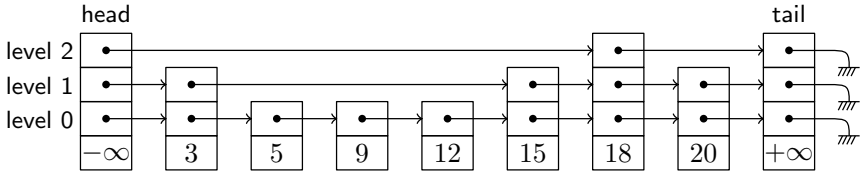
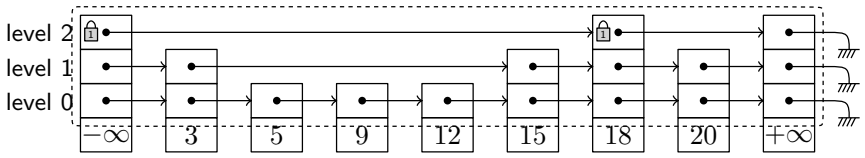
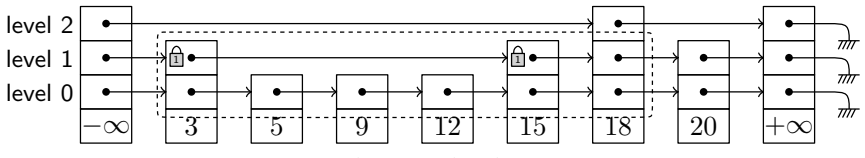


Figure 3.4: An example of skiplist

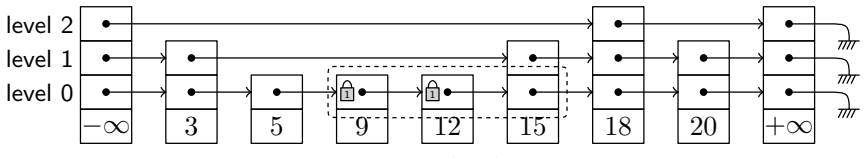
T_1 begins by grabbing the lock at level 2 on node $-\infty$ and node 18, as shown in Fig. 3.5(a). As 18 is beyond the position where 14 should be inserted, it decides then to go a level below. Afterwards, the algorithm proceeds as depicted in Fig. 3.5(b) and 3.5(c). In the figure, we use a dashed line to denote the



(a) initial locking by T_1



(b) T_1 on level 1

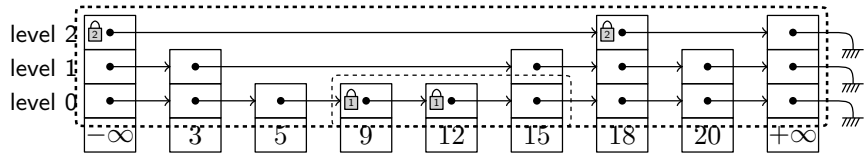


(c) T_1 on level 0

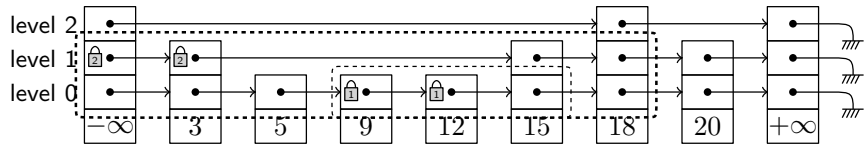
Figure 3.5: Progress of T_1 towards insertion of value 14

region of the skiplist that can potentially be modified by T_1, i.e., the region limited by $L[T_1]$, $U[T_1]$ and $H[T_1]$.

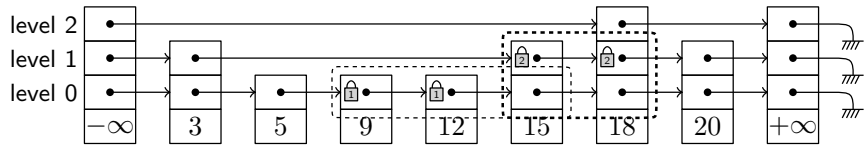
At this moment, T_2 starts its execution. Fig. 3.6 shows the progress made by T_2 toward the insertion of a level 2 node with value 16. As before, in this case the heavily dashed region represents the masked region of the skiplist that can be modified by T_2.



(a) initial locking by T_2



(b) T_2 on level 1



(c) final position of T_2 on level 1

Figure 3.6: Progress of T_2 towards insertion of value 16

Notice that the potentially modifiable regions by thread T_1 and T_2 intersect, as shown in Fig. 3.6(c). In this case, it is quite easy to see that under the assumption of weak-fairness, if T_2 continuously perform the same insertion, it can prevent T_1 from progressing. However, no problem exists under the assumption of strong fairness, since T_1 is not continuously enabled. In fact, it is quite easy to see that T_1 becomes disabled every time T_2 gets the lock at level 0 on node 15. ∗

A correct implementation of the SEARCH, INSERT and REMOVE operations is shown as Algorithms 3.7, 3.8 and 3.9 respectively. We call these implementations *pessimistic*, since each thread keeps a lock a couple of levels above its current position, just to ensure that nothing bad happens (as the situation described in Example 3.2). In the pessimistic implementations, we extend the skiplist data structure with a ghost masked region field r. Then, if sl is a skiplist, we use $sl.r$ to refer to this new field. $sl.r$ keeps the masked region made by all the nodes that form the skiplist sl.

As we said before, boxes denote ghost code added to aid the verification. In this particular case, the ghost variable m_r stores a masked region containing all the nodes and fields currently locked by the running thread. The set operations $\cup$ and $-$ are used for the manipulation of the corresponding sets of pairs. On the other hand, as we said, arrays L and U are used to keep an upper and lower bound of node's addresses that can be potentially modified by the thread performing the INSERT or REMOVE actions. In the case of SEARCH, these values denote the limits within the skiplist where the element to be searched can be located. Similarly, array H describes the upper bound for the levels involved in the operation.

The reader should convince himself that it is possible to show that thanks to the order in which locks are taken and released, a node cannot be removed until it has not been fully inserted. Similarly, it can be proved that if an element of the skiplist is being removed, then any INSERT program trying to insert a node with the same value into the skiplist will be prevented from progressing until the old node has been fully removed.

Algorithm 3.7 Pessimistic implementation of concurrent skiplist operation SEARCH

```
1:  procedure SEARCH(SkipList sl, Value v)
2:      int i := K − 1                                   | mrgn m_r := ∅
3:      Node* pred := sl.head                            | L := L{me ← sl.head}
                                                         | U := U{me ← sl.tail}
                                                         | H := H{me ← K − 1}
4:      pred.locks[i].lock()                             | m_r := m_r ∪ {(pred, i)}
5:      Node* curr := pred.next[i]
6:      curr.locks[i].lock()                             | m_r := m_r ∪ {(curr, i)}
7:      Node* cover
8:      while 0 ≤ i ∧ curr.val ≠ v do
9:          if i < K − 1 then
10:             pred.locks[i].lock()                     | m_r := m_r ∪ {(pred, i)}
                                                         | U := U{me ← pred.next[i + 1]}
11:             pred.locks[i + 1].unlock()               | m_r := m_r − {(pred, i + 1)}
                                                         | H := H{me ← i}
12:             if i < K − 2 then
13:                 cover.locks[i + 2].unlock()
14:             end if
15:             cover := curr
16:             curr := pred.next[i]
17:             curr.locks[i].lock()                     | m_r := m_r ∪ {(curr, i)}
18:         end if
19:         while curr.val < v do
20:             pred.locks[i].unlock()                   | m_r := m_r − {(pred, i)}
                                                         | L := L{me ← curr}
21:             pred := curr
22:             curr := pred.next[i]
23:             curr.locks[i].lock()                     | m_r := m_r ∪ {(curr, i)}
24:         end while
25:         i := i − 1
26:     end while
27:     Bool valueIsIn := (curr.val = v)
28:     if i = K − 1 then
29:         curr.locks[i].unlock()                       | m_r := m_r − {(curr, i)}
                                                         | U := U{me ← L[me]}
30:         pred.locks[i].unlock()                       | m_r := m_r − {(pred, i)}
                                                         | L := L{me ← null}
31:     else
32:         if i < K − 2 then
33:             cover.locks[i + 2].unlock()              | m_r := m_r − {(cover, i + 2)}
34:         end if
35:         curr.locks[i + 1].unlock()                   | m_r := m_r − {(curr, i + 1)}
                                                         | U := U{me ← L[me]}
36:         pred.locks[i + 1].unlock()                   | m_r := m_r − {(pred, i + 1)}
                                                         | L := L{me ← null}
37:     end if
38:     return valueIsIn
39: end procedure
```

Algorithm 3.8 Pessimistic implementation of concurrent skiplist operation INSERT

```
1:  procedure INSERT(SkipList sl, Value newval)
2:      Vector⟨Node*⟩upd[0..K − 1]
3:      lvl := randomLevel(K)
4:      Node* pred := sl.head
5:      pred.locks[K − 1].lock()
6:      Node* curr := pred.next[K − 1]
7:      curr.locks[K − 1].lock()
8:      Node* cover
9:      for i := K − 1 downto 0 do
10:         if i < K − 1 then
11:             pred.locks[i].lock()
12:             if i ≥ lvl then
13:                 pred.locks[i + 1].unlock()
14:             end if
15:             if i < K − 2 ∧ i > lvl − 2 then
16:                 cover.locks[i + 2].unlock()
17:             end if
18:             cover := curr
19:             curr := pred.next[i]
20:             curr.locks[i].lock()
21:         end if
22:         while curr.val < newval do
23:             pred.locks[i].unlock()
24:             pred := curr
25:             curr := pred.next[i]
26:             curr.locks[i].lock()
27:         end while
28:         upd[i] := pred
29:     end for
30:     if i < K − 2 then
31:         cover.locks[i + 2].unlock()
32:     end if
33:     Bool valueWasIn := (curr.val = newval)
34:     if valueWasIn then
35:         for i := lvl to 0 do
36:             upd[i].next[i].locks[i].unlock()
37:             upd[i].locks[i].unlock()
38:         end for
39:     else
40:         x := CreateNode(lvl, newval)
41:         for i := 0 to lvl do
42:             x.next[i] := upd[i].next[i]
43:             upd[i].next[i] := x
44:             x.next[i].locks[i].unlock()
45:             upd[i].locks[i].unlock()
46:         end for
47:     end if
48:     return ¬valueWasIn
49: end procedure
```

```
mrgn m_r := ∅
L := L{me ← sl.head}
U := U{me ← sl.tail}
H := H{me ← K − 1}
```

```
m_r := m_r ∪ {(pred, K − 1)}
```

```
m_r := m_r ∪ {(curr, K − 1)}
```

```
m_r := m_r ∪ {(pred, i)}
U := U{me ← pred.next[i + 1]}
```

```
m_r := m_r − {(pred, i + 1)}
H := H{me ← i}
```

```
m_r := m_r ∪ {(curr, i)}
```

```
m_r := m_r − {(pred, i)}
if (i ≥ lvl){
  L := L{me ← curr}
}
```

```
m_r := m_r ∪ {(curr, i)}
```

```
m_r := m_r − {(cover, i + 2)}
```

```
m_r := m_r − {(upd[i].next[i], i)}
```

```
m_r := m_r − {(upd[i], i)}
```

```
sl.r := sl.r ∪ {(x, i)}
```

```
m_r := m_r − {(x.next[i], i)}
```

```
m_r := m_r − {(upd[i], i)}
```

```
L := L{me ← null}
```

Algorithm 3.9 Pessimistic implementation of concurrent skiplist operation REMOVE

```
1:  procedure REMOVE(SkipList sl, Value v)
2:      Vector < Node* > upd[0..K − 1]
3:      Node* pred := sl.head
4:      pred.locks[K − 1].lock()
5:      Node* curr := pred.next[K − 1]
6:      curr.locks[K − 1].lock()
7:      cover := sl.tail
8:      deleteFrom := K − 1
9:      for i := K − 1 downto 0 do
10:         if i < K − 1 then
11:             pred.locks[i].lock()
12:             if pred.next[i + 1].val ≠ v then
13:                 deleteFrom := i
14:                 pred.locks[i + 1].unlock()
15:             end if
16:             if i < K − 2 ∧ cover.val ≠ v then
17:                 cover.locks[i + 2].unlock()
18:             end if
19:             cover := curr
20:             curr := pred.next[i]
21:             curr.locks[i].lock()
22:         end if
23:         while curr.val < v do
24:             pred.locks[i].unlock()
25:             pred := curr
26:             curr := pred.next[i]
27:             curr.locks[i].lock()
28:         end while
29:         upd[i] := pred
30:     end for
31:     if i < K − 2 then
32:         cover.locks[i + 2].unlock()
33:     end if
34:     Bool valueWasIn := (curr.val = v)
35:     for i := deleteFrom downto 0 do
36:         if upd[i].next[i] = curr ∧ curr.val = v then
37:             upd[i].next[i] := curr.next[i]
38:             curr.locks[i].unlock()
39:         else
40:             upd[i].next[i].locks[i].unlock()
41:         end if
42:         upd[i].locks[i].unlock()
43:     end for
44:     if valueWasIn then
45:         free (curr)
46:     end if
47:     return valueWasIn
48: end procedure
```

```
mrgn m_r := ∅
L := L{me ← sl.head}
U := U{me ← sl.tail}
H := H{me ← K − 1}
```

```
m_r := m_r ∪ {(pred, K − 1)}
```

```
m_r := m_r ∪ {(curr, K − 1)}
```

```
m_r := m_r ∪ {(pred, i)}
U := U{me ← curr}
```

```
m_r := m_r − {(pred, i + 1)}
H := H{me ← i}
```

```
m_r := m_r ∪ {(curr, i)}
```

```
m_r := m_r − {(pred, i)}
if (cover.val ≠ e){
   L := L{me ← curr}
}
```

```
m_r := m_r ∪ {(curr, i)}
```

```
sl.r := sl.r − {(curr, i)}
```

```
m_r := m_r − {(curr, i)}
```

```
m_r := m_r − {(upd[i].next[i], i)}
```

```
m_r := m_r − {(upd[i], i)}
L := L{me ← null}
```

Example 3.3 (Skiplist shape preservation)
Let *sl* be a pointer to an instance of the class *SkipList* described before. The following predicate captures whether *sl* points to a well-formed skiplist of height 4 or less:

$$SkipList_4(h, head, tail) \triangleq OList(h, head, 0) \wedge \tag{3.2}$$

$$\begin{pmatrix} tail.next[0] = null \wedge tail.next[1] = null \\ tail.next[2] = null \wedge tail.next[3] = null \end{pmatrix} \wedge \tag{3.3}$$

$$\begin{pmatrix} SubList(h, head, tail, 1, head, tail, 0) \wedge \\ SubList(h, head, tail, 2, head, tail, 1) \wedge \\ SubList(h, head, tail, 3, head, tail, 2) \end{pmatrix} \tag{3.4}$$

Assuming that *head* (resp. *tail*) points to the first (resp. last) node of a skiplist, predicate $SkipList_4$ says that in fact between both nodes there is a structure with the shape of a skiplist of height 4. Considering the definition of $SkipList_4$, the predicate *OList* in (3.2) describes that in heap *h*, the pointer *head* points to an ordered linked-lists if we repeatedly follow the pointers at level 0. The predicate (3.3) indicates that all levels are *null* terminated, and (3.4) indicates that each level is in fact a sublist of its nearest lower level. Predicates of this kind also allow to express the effect of program statements via first order transition relations. ∗

Example 3.4 (Transition relation on skiplist INSERT operation)
Consider the statement at line 43 in program INSERT shown as Algorithm 3.8 on a skiplist of height 4, taken by thread with id *j*. This transition corresponds to a new node $x^{[j]}$ at level *i* being connected to the skiplist, as depicted in Fig. 3.7.

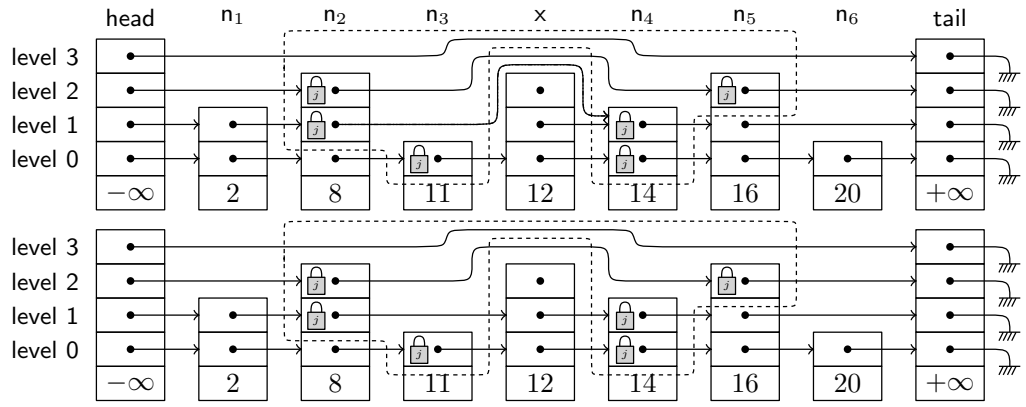


Figure 3.7: Skiplist modification at line 43 before and after inserting value 12 with $i = 1$

For the sake of simplicity, from this point on we use *x* to represent $x^{[j]}$. If the memory layout from pointer *sl* is that of a skiplist before the statement at line 43 is executed, then it is also a skiplist after the execution:

$$SkipList_4(h, sl.head, sl.tail) \wedge \varphi_{aux} \wedge \rho_{43}^{[j]}(V, V') \rightarrow SkipList_4(h', sl'.head, sl', tail)$$

The effect of the statement at line 43 is represented by the first-order transition relation $\rho_{43}^{[t]}$. To ensure this property, *i* is required to be a valid level, and the key of the nodes that will be pointing to *x* must be lower than the key of node *x*. Moreover, the masked region of locked nodes remains unchanged. Predicate φ_{aux} contains support invariants. For simplicity, we use *prev* for $upd^{[j]}[i]$. Then, the full verification condition is $SkipList_4(h, sl.head, sl.tail) \wedge \phi \rightarrow SkipList_4(h', sl'.head, sl'.tail)$, where ϕ is:

$$\left(\begin{array}{rcll} x.key & = & newval & \wedge \\ prev.key & < & newval & \wedge \\ x.next[i].key & > & newval & \wedge \\ prev.next[i] & = & x.next[i] & \wedge \\ (x,i) \notin sl.r \wedge 0 \leq i \leq 3 & & & \end{array}\right) \wedge \left(\begin{array}{cl} at_{43}^{[j]} & \wedge \\ prev'.next[i] = x & \wedge \\ at_{44}^{\prime[j]} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ x' = x & \ldots \end{array}\right) \quad *$$

Note that in the previous verification condition we do not explicitly indicate in all cases that the rest of the nodes, memory and local variables of other threads remain unchanged. This preservation predicates are generated as part of a verification condition based on those elements of the formula that are not modified by the statement. A full example of a verification condition is given in Chapter 5, where the decision procedure for concurrent skiplists is described. The examples we have seen should be enough to illustrate that to be able to automatically prove VCs for the verification of skiplist manipulating algorithms, we require a theory that allows to reason about heaps, addresses, nodes, masked regions, ordered lists and sublists.

4

TLL3: A Decision Procedure for Concurrent Lock-Coupling Lists

The automatic check of the proof represented by a verification diagram requires decision procedures to verify the generated verification conditions. These decision procedures must deal with formulas containing terms belonging to different theories. In particular, for concurrent lists the decision procedure must reason about pointer data structures with a list layout, regions and locks. To obtain a suitable decision procedure, we extend the Theory of Linked Lists (TLL) [RZ06a], a decidable theory including reachability of list-like structures. However, this theory lacks the expressivity to describe locked lists of cells, a fundamental component in our proofs. From the extension of TLL we get TLL3 [SS10]. In this chapter we focus on the definition, description and analysis of TLL3 as well as a decision procedure for this theory.

We begin with a brief description of the basic notation and concepts. A signature Σ is a triple (S, F, P) where S is a set of sorts, F is a set of function symbols and P is a set of predicate symbols constructed using the sorts in S. If $\Sigma_1 = (S_1, F_1, P_1)$ and $\Sigma_2 = (S_2, F_2, P_2)$ are two signatures, we define their union $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$. Similarly we say that $\Sigma_1 \subseteq \Sigma_2$ when $S_1 \subseteq S_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$.

Given a signature Σ, we assume the standard notions of Σ-term, Σ-literal and Σ-formula. A literal is flat if it is of the form $x = y$, $x \neq y$, $x = f(y_1, \ldots, y_n)$, $p(y_1, \ldots, y_n)$ or $\neg p(y_1, \ldots, y_n)$, where $x, y, y_1, \ldots, y_n$ are variables, f is a function symbol and p is a predicate symbol. If $t(\varphi)$ is a term (resp. formula), then we denote with $V_\sigma(t)$ (resp. $V_\sigma(\varphi)$) the set of variables of sort σ occurring in t (resp. φ).

We assume the usual notion of Σ-interpretation over a set V of variables as a map which assigns a value to each symbol in Σ and V. Let $\mathcal{A}$ be a Σ-interpretation over V. Then, for each $s \in S$, $\mathcal{A}_s$ is a set of elements called the domain of s; for each symbol $f : s_1 \times \cdots \times s_n \to s$ in F, $\mathcal{A}_f$ is a function that goes from $\mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n}$ to elements in $\mathcal{A}_s$ and for each symbol $p : s_1 \times \cdots \times s_n$ in P, $\mathcal{A}_p$ is a relation over elements of $\mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n}$.

A Σ-structure is a Σ-interpretation over an empty set of variables. A Σ-formula over a set V of variables is satisfiable whenever it is true in some Σ-interpretation over V. Let Ω be a signature, $\mathcal{A}$ a Ω-interpretation over a set V of variables, $\Sigma \subseteq \Omega$ and $U \subseteq V$. $\mathcal{A}^{\Sigma,U}$ denotes the interpretation obtained from $\mathcal{A}$ restricting it to interpret only the symbols in Σ and the variables in U. We use $\mathcal{A}^{\Sigma}$ to denote $\mathcal{A}^{\Sigma,\emptyset}$. A Σ-theory is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class of Σ-structures. Given a theory $T = (\Sigma, \mathbf{A})$, a T-interpretation is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}^{\Sigma} \in \mathbf{A}$. Given a Σ-theory T, a Σ-formula φ over a set of variables V is T-satisfiable if it is true on a T-interpretation over V.

Signature:	Σ_{cell}		
Sorts:	cell, elem, addr, thid		
Functions:	*error*	:	cell
	mkcell	:	elem $\times$ addr $\times$ thid $\rightarrow$ cell
	_.*data*	:	cell $\rightarrow$ elem
	_.*next*	:	cell $\rightarrow$ addr
	_.*lockid*	:	cell $\rightarrow$ thid
	_.*lock*	:	cell $\rightarrow$ thid $\rightarrow$ cell
	_.*unlock*	:	cell $\rightarrow$ cell
Signature:	Σ_{mem}		
Sorts:	mem, addr, cell		
Functions:	*null*	:	addr
	_[_]	:	mem $\times$ addr $\rightarrow$ cell
	upd	:	mem $\times$ addr $\times$ cell $\rightarrow$ mem
Signature:	Σ_{reach}		
Sorts:	mem, addr, path		
Functions:	ϵ	:	path
	[_]	:	addr $\rightarrow$ path
Predicates:	*append*	:	path $\times$ path $\times$ path
	reach	:	mem $\times$ addr $\times$ addr $\times$ path
Signature:	Σ_{set}		
Sorts:	addr, set		
Functions:	$\emptyset$	:	set
	{ _ }	:	addr $\rightarrow$ set
	$\cup, \cap, \setminus$	:	set $\times$ set $\rightarrow$ set
Predicates:	$\in$	:	addr $\times$ set
	$\subseteq$	:	set $\times$ set
Signature:	Σ_{settid}		
Sorts:	thid, settid		
Functions:	$\emptyset_{\text{T}}$	:	settid
	$\{\_\}_{\text{T}}$	:	thid $\rightarrow$ settid
	$\cup_{\text{T}}, \cap_{\text{T}}, \setminus_{\text{T}}$	:	settid $\times$ settid $\rightarrow$ settid
Predicates:	$\in_{\text{T}}$	:	thid $\times$ settid
	$\subseteq_{\text{T}}$	:	settid $\times$ settid
Signature:	Σ_{bridge}		
Sorts:	mem, addr, set, path		
Functions:	*path2set*	:	path $\rightarrow$ set
	addr2set	:	mem $\times$ addr $\rightarrow$ set
	getp	:	mem $\times$ addr $\times$ addr $\rightarrow$ path
	fstlock	:	mem $\times$ path $\rightarrow$ addr

Figure 4.1: The signature of the TLL3 theory

signature: Σ_{cell}

interpretation:

- $mkcell^{\mathcal{A}}(e,a,k) = \langle e,a,k\rangle$
- $\langle e,a,t\rangle.data^{\mathcal{A}} = e$
- $\langle e,a,t\rangle.next^{\mathcal{A}} = a$
- $\langle e,a,t\rangle.lockid^{\mathcal{A}} = t$
- $\langle e,a,t\rangle.lock^{\mathcal{A}}(t') = \langle e,a,t'\rangle$
- $\langle e,a,t\rangle.unlock^{\mathcal{A}} = \langle e,a,\oslash\rangle$
- $error^{\mathcal{A}}.next^{\mathcal{A}} = null^{\mathcal{A}}$

for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$, $k \in \mathcal{A}_{\mathsf{thid}}$ and $t, t' \in \mathcal{A}_{\mathsf{thid}}$

Signature: Σ_{mem}

Interpretation:

- $m[a]^{\mathcal{A}} = m(a)$
- $upd^{\mathcal{A}}(m,a,c) = m_{a\mapsto c}$
- $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$

for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$

Signature: Σ_{reach}

Interpretation:

- $\epsilon^{\mathcal{A}}$ is the empty sequence
- $[i]^{\mathcal{A}}$ is the sequence containing $i \in \mathcal{A}_{\mathsf{addr}}$ as the only element
- $([i_1,\ldots,i_n],[j_1,\ldots,j_m],[i_1,\ldots,i_n,j_1,\ldots,j_m]) \in append^{\mathcal{A}}$ iff
  i_k and j_l are all distinct
- $(m,i,j,p) \in reach^{\mathcal{A}}$ iff
  - $i = j$ and $p = \epsilon$, or
  - there exist addresses $i_1,\ldots,i_n \in \mathcal{A}_{\mathsf{addr}}$ such that:
    (a) $p = [i_1,\ldots,i_n]$ (c) $m(i_r).next^{\mathcal{A}} = i_{r+1}$, for $1 \le r < n$
    (b) $i_1 = i$ (d) $m(i_n).next^{\mathcal{A}} = j$

Signature: Σ_{set}

Interpretation:

- The symbols $\emptyset$, $\{\_\}$, $\cup$, $\cap$, $\setminus$, $\in$ and $\subseteq$ are interpreted according to their standard interpretation over sets of addresses.

Signature: Σ_{settid}

Interpretation:

- The symbols $\emptyset_{\mathrm{T}}$, $\{\_\}_{\mathrm{T}}$, $\cup_{\mathrm{T}}$, $\cap_{\mathrm{T}}$, $\setminus_{\mathrm{T}}$, $\in_{\mathrm{T}}$ and $\subseteq_{\mathrm{T}}$ are interpreted according to their standard interpretation over sets of thread identifiers.

Signature: Σ_{bridge}

Interpretation:

- $addr2set^{\mathcal{A}}(m,i) = \left\{ j \in \mathcal{A}_{\mathsf{addr}} \mid \exists p \in \mathcal{A}_{\mathsf{path}} \text{ s.t. } (m,i,j,p) \in reach \right\}$
- $path2set^{\mathcal{A}}(p) = \{i_1,\ldots,i_n\}$ for $p = [i_1,\ldots,i_n] \in \mathcal{A}_{\mathsf{path}}$
- $getp^{\mathcal{A}}(m,i,j) = \begin{cases} p & \text{if } (m,i,j,p) \in reach^{\mathcal{A}} \\ \epsilon & \text{otherwise} \end{cases}$

  for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$ and $i, j \in \mathcal{A}_{\mathsf{addr}}$
- $fstlock^{\mathcal{A}}(m,[a_1,\ldots,a_n]) = \begin{cases} a_k & \text{if there is } 1 \le k \le n \text{ such that} \\ & \text{for all } 1 \le j < k, m[a_j].lockid = \oslash \\ & \text{and } m[a_k].lockid \neq \oslash \\ null & \text{otherwise} \end{cases}$

  for each $m \in \mathcal{A}_{\mathsf{mem}}$ and $a_1,\ldots a_n \in \mathcal{A}_{\mathsf{addr}}$

Figure 4.2: Characterization of a TLL3-interpretation $\mathcal{A}$

Formally, the theory of linked lists is defined as $\mathsf{TLL} = (\Sigma_{\mathsf{TLL}}, \mathbf{TLL})$, where

$$\Sigma_{\mathsf{TLL}} \;:=\; \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{reach}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{bridge}}$$

The sorts, functions and predicates belonging to these signatures are described as part of Fig. 4.1, while **TLL** is the class of Σ_{TLL}-structures satisfying the conditions shown in Fig. 4.2. In fact, these figures corresponds to the signatures and interpretations for TLL3 while TLL is contained into TLL3, as we will see next.

We extend TLL into the theory for concurrent single linked lists with locks $\mathsf{TLL3} := (\Sigma_{\mathsf{TLL3}}, \mathbf{TLL3})$, where

$$\Sigma_{\mathsf{TLL3}} = \Sigma_{\mathsf{TLL}} \cup \Sigma_{\mathsf{settid}} \cup \{lockid, lock, unlock, fstlock\}$$

Each sort σ in a Σ_{TLL3}-structure is mapped to a non-empty set $\mathcal{A}_\sigma$ such that:

(a) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{thid}}$

(b) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$

(c) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$

(d) $\mathcal{A}_{\mathsf{set}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$

(e) $\mathcal{A}_{\mathsf{settid}}$ is the power-set of $\mathcal{A}_{\mathsf{thid}}$

The sorts, functions and predicates of TLL3 correspond to the signatures shown in Fig. 4.1 and the interpretation of each function and predicate is depicted in Fig. 4.2. Informally, Σ_{cell} models *cells*, structures containing an element (data), an address (pointer) and a lock owner, which represents a node in a linked list. Σ_{mem} models the memory. Σ_{reach} models finite sequences of non-repeating addresses, to represent paths. Σ_{set} models sets of addresses. Finally, Σ_{bridge} is a *bridge theory* containing auxiliary functions that map paths of addresses to set of addresses or let us obtain the set of addresses reachable from a given address following a chain of *next* fields. The sort thid contains thread identifiers. The sorts addr, elem and thid are uninterpreted, except that $\oslash$: thid is different from all others thread ids. Otherwise, $\Sigma_{\mathsf{addr}} = (\mathsf{addr}, \emptyset, \emptyset)$, $\Sigma_{\mathsf{elem}} = (\mathsf{elem}, \emptyset, \emptyset)$ and $\Sigma_{\mathsf{thid}} = (\mathsf{thid}, \emptyset, \emptyset)$.

Example 4.1
Consider the list shown in Fig. 3.1 in Section 3.1. For that particular list and assuming that nodes are locked by let's say T_1, we can construct a model $\mathcal{A}$ such that for instance:

$$\begin{aligned}
\mathcal{A}_{\mathsf{addr}} &= \{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07\} \\
\mathcal{A}_{\mathsf{ord}=}\mathcal{A}_{\mathsf{elem}} &= \{-\infty, 3, 5, 9, 10, 17, +\infty\} \\
\mathcal{A}_{\mathsf{thid}} &= \{T_1, T_2, \oslash\} \\
\mathcal{A}_{\mathsf{cell}} &= \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{ord}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{thid}} \\
\mathcal{A}_{\mathsf{mem}} &= \{m : \mathcal{A}_{\mathsf{addr}} \to \mathcal{A}_{\mathsf{cell}}\}
\end{aligned}$$

where

$$\begin{aligned}
null^{\mathcal{A}} &= 0x00 \\
error^{\mathcal{A}} &= \langle\ -\infty\ ,\ -\infty\ ,\ null\ ,\ \oslash\ \rangle
\end{aligned}$$

$$\begin{aligned}
m(0x00) &= \langle\ -\infty\ ,\ -\infty\ ,\ null\ ,\ \oslash\ \rangle \\
m(0x01) &= \langle\ -\infty\ ,\ +\infty\ ,\ null\ ,\ \oslash\ \rangle \\
m(0x02) &= \langle\ 3\ ,\ 3\ ,\ 0x03\ ,\ \oslash\ \rangle \\
m(0x03) &= \langle\ 5\ ,\ 5\ ,\ 0x04\ ,\ T_1\ \rangle \\
m(0x04) &= \langle\ 9\ ,\ 9\ ,\ 0x05\ ,\ T_1\ \rangle \\
m(0x05) &= \langle\ 10\ ,\ 10\ ,\ 0x06\ ,\ \oslash\ \rangle \\
m(0x06) &= \langle\ 17\ ,\ 17\ ,\ 0x07\ ,\ \oslash\ \rangle \\
m(0x07) &= \langle\ +\infty\ ,\ +\infty\ ,\ 0x00\ ,\ \oslash\ \rangle
\end{aligned}$$

*

We are interested in analyzing the satisfiability of quantifier-free first order formulas. Hence, if φ is a formula, we first write it into its disjunctive normal form, let's say $\varphi_1 \vee \cdots \vee \varphi_n$. And then, we just verify the satisfiability of any of the φ_i, where each φ_i is a conjunction of TLL3 literals. We now classify the TLL3 literals into normalized and non-normalized ones. Non-normalized literals have the property that they can be written in terms of normalized ones. We now define the set of normalized TLL3-literals.

Definition 4.1 (TLL3-normalized literals).
A TLL3-literal is normalized if it is a flat literal of the form:

$e_1 \neq e_2$	$a_1 \neq a_2$	
$a = null$	$c = error$	
$c = mkcell(e, a, t)$	$c = rd(m, a)$	$m_2 = upd(m_1, a, c)$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$p_1 \neq p_2$	$p = [a]$	$p_1 = rev(p_2)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$
$s = addr2set(m, a)$	$p = getp(m, a_1, a_2)$	
$t_1 \neq t_2$	$a = fstlock\,(m, p)$	

where e, e_1 and e_2 are elem*-variables, a, a_1 and a_2 are* addr*-variables, c is a* cell*-variable, m, m_1 and m_2 are* mem*-variables, p, p_1, p_2 and p_3 are* path*-variables, and t, t_1 and t_2 are* thid*-variables.* †

The remaining literals that are part of TLL3 but not detailed in the previous definition, can be written in terms of normalized literals considering the following equivalences:

$$
\begin{aligned}
t = c.lockid &\leftrightarrow (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a)\,[c = mkcell\,(e, a, t)] \\
c_1 = c_2.lock\,(t) &\leftrightarrow c_2.data = c_1.data \wedge c_2.next = c_1.next \wedge t = c_1.lockid \\
c_1 = c_2.unlock &\leftrightarrow c_2.data = c_1.data \wedge c_2.next = c_1.next \wedge \oslash = c_1.lockid \\
c_1 \neq_{\mathsf{cell}} c_2 &\leftrightarrow c_1.data \neq c_2.data \vee c_1.next \neq c_2.next \vee c_1.lockid \neq c_2.lockid \\
m_1 \neq_{\mathsf{mem}} m_2 &\leftrightarrow (\exists_{\mathsf{addr}} a)\,[rd(m_1, a) \neq rd(m_2, a)] \\
s_1 \neq s_2 &\leftrightarrow (\exists_{\mathsf{addr}} a)\,[a \in (s_1 \setminus s_2) \cup (s_2 \setminus s_1)] \\
s = \emptyset &\leftrightarrow s = s \setminus s \\
s_3 = s_1 \cap s_2 &\leftrightarrow s_3 = (s_1 \cup s_2) \setminus ((s_1 \setminus s_2) \cup (s_2 \setminus s_1)) \\
a \in s &\leftrightarrow \{a\} \subseteq s \\
s_1 \subseteq s_2 &\leftrightarrow s_2 = s_1 \cup s_2 \\
p = \epsilon &\leftrightarrow append(p, p, p) \\
reach(m, a_1, a_2, p) &\leftrightarrow a_2 \in addr2set(m, a_1) \wedge p = getp(m, a_1, a_2)
\end{aligned}
$$

this means that we can rewrite such literals using the following table:

Flat:	$t = c.lockid$
Normalized:	$c = mkcell\,(e, a, t)$
Proviso:	e and a are fresh variables.
Flat:	$c_1 = c_2.lock\,(t)$
Normalized:	$c_1 = mkcell\,(e, a, t) \wedge c_2 = mkcell\,(e, a, t')$
Proviso:	e, a and t' are fresh variables.
Flat:	$c_1 = c_2.lock\,(t)$
Normalized:	$c_1 = mkcell\,(e, a, t) \wedge c_2 = mkcell\,(e, a, t')$
Proviso:	e, a and t' are fresh variables.
Flat:	$c_1 = c_2.unlock$
Normalized:	$c_1 = mkcell\,(e, a, \oslash) \wedge c_2 = mkcell\,(e, a, t')$
Proviso:	e, a and t' are fresh variables.

Flat:	$c_1 \neq c_2$
Normalized:	$c_1.data \neq c_2.data \vee c_1.next \neq c_2.nextc_1.lockid \neq c_2.lockid$
Proviso:	–
Flat:	$m_1 \neq m_2$
Normalized:	$m[a] \neq m[a]$
Proviso:	a is fresh.
Flat:	$s_1 \neq s_2$
Normalized:	$s_{12} = s_1 \setminus s_2 \wedge s_{21} = s_2 \setminus s_1 \wedge s_3 = s_{12} \cup s_{21} \wedge s = s_3 \cup \{a\} \wedge \{a\} \subseteq s$
Proviso:	s_{12}, s_{21}, s_3, s and a are fresh.
Flat:	$s = \emptyset$
Normalized:	$s = s \setminus s$
Proviso:	–
Flat:	$s_3 = s_1 \cap s_2$
Normalized:	$s_{12} = s_1 \setminus s_2 \wedge s_{21} = s_2 \setminus s_1 \wedge s_{u_1} = s_1 \cup s_2 \wedge s_{u_2} = s_{12} \cup s_{21} \wedge s_3 = s_{u_1} \setminus s_{u_2}$
Proviso:	s_{12}, s_{21}, s_{u_1} and s_{u_2} are fresh.
Flat:	$a \in s$
Normalized:	$s = \{a\} \cup s$
Proviso:	–
Flat:	$s_1 \subseteq s_2$
Normalized:	$s_2 = s_1 \cup s_2$
Proviso:	–
Flat:	$p = \epsilon$
Normalized:	$append(p, p, p)$
Proviso:	-
Flat:	$reach(m, a_1, a_2, p)$
Normalized:	$a_2 \in addr2set(m, a_1) \wedge p = getp(m, a_1, a_2)$
Proviso:	-

4.1 Decidability of TLL3

TLL [RZ06a] enjoys the finite model property. We now show that TLL3 also has the finite model property with respect to domains elem, addr and thid. Hence, TLL3 is decidable because one can enumerate Σ_{TLL3}-structures up to a certain cardinality. Notice that a bound on the domain of these sorts it is enough to get finite interpretations for the remaining sorts (cell, mem, path, set and settid) as the elements in the domains of these latter sorts are constructed using the elements in the domains of elem, addr and thid.

Definition 4.2 (Finite Model Property).
Let $\Sigma = (S, F, P)$ be a signature, $S_0 \subseteq S$ be a set of sorts, and T be a Σ-theory. T has the finite model property with respect to S_0 if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ such that for each sort $\sigma \in S_0$, $\mathcal{A}_\sigma$ is finite. †

Lemma 4.1:
Deciding the TLL3-satisfiability of a quantifier-free TLL3-formula is equivalent to verifying the TLL3-satisfiability of the normalized TLL3-literals. ♠

Proof The proof is by cases on the shape of all possible TLL3-literals. Let φ be a quantifier-free TLL3 formula. As we are interested in the satisfiability problem, φ can be reduced to its disjunctive normal form $\varphi_1 \vee \cdots \vee \varphi_n$, leading us to the problem of checking whether some φ_i is satisfiable. Each φ_i is a

conjunction of TLL3 normalized and non-normalized literals. Then, it just remains to see that if a model satisfies a non-normalized literal, then it also satisfies its representation made by normalized literals.

For instance, let's consider the non-normalized literal $c_1 = c_2.lock(t)$. According to the table we presented before, this literal can be written as a conjunction of normalized literals of the form $c_1 = mkcell(e, a, t) \wedge c_2 = mkcell(e, a, t')$ where e, a and t' are fresh variables. Let $\mathcal{A}$ be a model satisfying $c_1 = c_2.lock(t)$. Then, we show that $\mathcal{A} \vDash c_1 = c_2.lock(t) \leftrightarrow \mathcal{A} \vDash c_1 = mkcell(e, a, t) \wedge c_2 = mkcell(e, a, t')$

$$
\begin{array}{lll}
\mathcal{A} \vDash & c_1 = c_2.lock(t) & \leftrightarrow \\
& c_1^{\mathcal{A}} = c_2^{\mathcal{A}}.lock^{\mathcal{A}}(t^{\mathcal{A}}) & \leftrightarrow \\
& c_1^{\mathcal{A}} = c_2^{\mathcal{A}}.lock^{\mathcal{A}}(t^{\mathcal{A}}) \wedge c_2{}^{\mathcal{A}} = \langle e^{\mathcal{A}}, a^{\mathcal{A}}, t'^{\mathcal{A}} \rangle & \leftrightarrow \\
& c_1^{\mathcal{A}} = \langle e^{\mathcal{A}}, a^{\mathcal{A}}, t^{\mathcal{A}} \rangle \wedge c_2{}^{\mathcal{A}} = \langle e^{\mathcal{A}}, a^{\mathcal{A}}, t'^{\mathcal{A}} \rangle & \leftrightarrow \\
& c_1^{\mathcal{A}} = mkcell^{\mathcal{A}}(e^{\mathcal{A}}, a^{\mathcal{A}}, t^{\mathcal{A}}) \wedge c_2^{\mathcal{A}} = mkcell^{\mathcal{A}}(e^{\mathcal{A}}, a^{\mathcal{A}}, t'^{\mathcal{A}}) & \leftrightarrow \\
\mathcal{A} \vDash & c_1 = mkcell(e, a, t) \wedge c_2 = mkcell(e, a, t') &
\end{array}
$$

The remaining cases can be proved in a similar way. □

Consider an arbitrary TLL3-interpretation $\mathcal{A}$ satisfying a conjunction of normalized TLL3-literals Γ. We show that if there are sets $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{thid}}$ (these sets are the domains interpreted by $\mathcal{A}$ for the sorts elem, addr and thid respectively) then there are finite sets $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$ and $\mathcal{B}_{\mathsf{thid}}$ with bounded cardinalities (the bound depending on Γ). $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$ and $\mathcal{B}_{\mathsf{thid}}$ can in turn be used to obtain a finite interpretation $\mathcal{B}$ satisfying Γ.

Before proceeding with the proof that TLL3 enjoys the finite model property, we define some auxiliary functions. We start by defining the function *first*. Let $X \subseteq \tilde{X}$, $m : \tilde{X} \to Z \times \tilde{X} \times Y$ and $a \in X$. The function $first(m, a, X)$ is defined by

$$
first(m, a, X) = \begin{cases}
null & \text{if } (\forall r \geq 1)\,[m^r(a).next \notin X] \\
m^s(a).next & \text{if } (\exists s \geq 1)\Big[m^s(a).next \in X \wedge \\
 & \qquad (\forall r \geq 1)\,(r < s \rightarrow m^r(a).next \notin X)\Big]
\end{cases}
$$

where $m^1(a).next$ stands for $m(a).next$ and $m^{n+1}(a).next$ for $m(m^n(a).next).next$ when $n > 1$.

Basically, given a set of addresses X, function *first* chooses the next address in X that can be reached from a given address following repeatedly the *next* pointer. It is easy to see, for example, that if $m(a).next \in X$ then $first(m, a, X) = m(a).next$. We will later filter out unnecessary intermediate nodes and use *first* to bypass properly the removed nodes, preserving the important connectivity properties.

Lemma 4.2:
Let $X \subseteq \tilde{X}$, $m_1, m_2 : \tilde{X} \to Z \times \tilde{X} \times Y$, $i, j \in X$, $c \in Z \times X \times Y$ and $i \neq j$. Then:

(a) If $m_1(i).next \in X$ then $first(m_1, i, X) = m_1(i).next$

(b) If $m_1 = upd(m_2, i, c)$ then $first(m_1, j, X) = first(m_2, j, X)$ ♠

Proof (a) immediate.

(b) Let $m_1 = upd(m_2, i, c)$ and assume first that $m_1^r(j).next \notin X$, for all $r \geq 1$. By induction it can be shown that $m_1^r(j) = m_2^r(j)$, for each $r \geq 1$. It follows that $first(m_1, j, X) = j$ and $first(m_2, j, X) = j$.

If instead $m_1^s(j).next \in X$, for some $s \geq 1$, assume without loss of generality that $first(m_1, j, X) = m_1^s(j).next$. By induction, it is possible to show that $m_1^r(j) = m_2^r(j)$, for each $1 \leq r \leq s$. It follows that $first(m_1, j, X) = m_1^s(j).next = m_2^b(j).next = first(m_2, j, X)$. □

We also define the *compress* function which, given a path p and a set X of addresses, returns the path obtained from p by removing all the addresses that do not belong to X.

$$compress([i_1,\ldots,i_n],X) = \begin{cases} \epsilon & \text{if } n=0 \\ [i_1]\circ compress([i_2,\ldots,i_n],X) & \text{if } n>0 \text{ and } i_1\in X \\ compress([i_2,\ldots,i_n],X) & \text{otherwise} \end{cases}$$

We conclude by defining the function *diseq* [RZ06a] that outputs a set of addresses accountable for disequality of two given paths:

$$diseq([i_1,\ldots,i_n],[j_1,\ldots j_m]) = \begin{cases} \emptyset & \text{if } n=m=0 \\ \{i_1\} & \text{if } n>0 \text{ and } m=0 \\ \{j_1\} & \text{if } n=0 \text{ and } m>0 \\ \{i_1,j_1\} & \text{if } n,m>0 \text{ and } i_1\neq j_1 \\ diseq([i_2,\ldots,i_m],[j_2,\ldots,j_m]) & \text{otherwise} \end{cases}$$

and function *common* [RZ06a] that outputs an element common to two paths (an element that witnesses that $path2set(p)\cap path2set(q)\neq\emptyset$):

$$common([i_1,\ldots,i_n],p) = \begin{cases} \emptyset & \text{if } n=0 \\ \{i_1\} & \text{if } n>0 \text{ and } i_1\in path2set(p) \\ common([i_2,\ldots,i_n],p) & \text{otherwise} \end{cases}$$

Lemma 4.3 (Finite Model Property):
Let Γ be a conjunction of normalized TLL3-literals. Let $\overline{e}=|V_{\mathsf{elem}}(\Gamma)|$, $\overline{a}=|V_{\mathsf{addr}}(\Gamma)|$, $\overline{m}=|V_{\mathsf{mem}}(\Gamma)|$, $\overline{p}=|V_{\mathsf{path}}(\Gamma)|$ and $\overline{t}=|V_{\mathsf{thid}}(\Gamma)|$. Then the following are equivalent:

1. *Γ is TLL3-satisfiable;*

2. *Γ is true in a TLL3 interpretation $\mathcal{B}$ such that*

$$|\mathcal{B}_{\mathsf{addr}}| \leq \overline{a}+1+\overline{m}\,\overline{a}+\overline{p}^2+\overline{p}^3$$
$$|\mathcal{B}_{\mathsf{thid}}| \leq \overline{t}+\overline{m}\,|\mathcal{B}_{\mathsf{addr}}|+1$$
$$|\mathcal{B}_{\mathsf{elem}}| \leq \overline{e}+\overline{m}\;|\mathcal{B}_{\mathsf{addr}}|$$

♠

Proof $(2\rightarrow 1)$. Immediate.

$(1\rightarrow 2)$. As TLL3 is an extension of TLL, some normalized literals are shared between both theories. Hence, here we limit ourselves to prove the implication for the new TLL3-literals only. The proof for normalized literals shared between TLL and TLL3 can be seen in [RZ06b].

Let now $\mathcal{A}$ be a TLL3-interpretation satisfying Γ. We will use $\mathcal{A}$ to construct a finite TLL3-interpretation $\mathcal{B}$ satisfying Γ.

$$
\begin{array}{rcll}
\mathcal{B}_{\mathsf{addr}} & = & V^{\mathcal{A}}_{\mathsf{addr}} \cup \left\{null^{\mathcal{A}}\right\} & \cup \\
& & \left\{m^{\mathcal{A}}(v^{\mathcal{A}}).next^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in V_{\mathsf{addr}}\right\} & \cup \\
& & \left\{v \in diseq(p^{\mathcal{A}}, q^{\mathcal{A}}) \mid \text{ the literal } p \neq q \text{ is in } \Gamma\right\} & \cup \\
& & \left\{v \in common(p_1{}^{\mathcal{A}}, p_2{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and } \right. & \\
& & \qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) \neq \emptyset\right\} & \cup \\
& & \left\{v \in common(p_1{}^{\mathcal{A}} \circ p_2{}^{\mathcal{A}}, p_3{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and } \right. & \\
& & \qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) = \emptyset\right\} &
\end{array}
$$

$$
\mathcal{B}_{\mathsf{thid}} = V^{\mathcal{A}}_{\mathsf{thid}} \cup \left\{ \oslash \right\} \cup \left\{m^{\mathcal{A}}(v).lockid^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in X\right\}
$$

$$
\mathcal{B}_{\mathsf{elem}} = V^{\mathcal{A}}_{\mathsf{elem}} \cup \left\{m^{\mathcal{A}}(v).data^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in X\right\}
$$

and let

$$
\begin{array}{rcll}
error^{\mathcal{B}} & = & error^{\mathcal{A}} & \\
null^{\mathcal{B}} & = & null^{\mathcal{A}} & \\
e^{\mathcal{B}} & = & e^{\mathcal{A}} & \text{for each } e \in V_{\mathsf{elem}} \\
a^{\mathcal{B}} & = & a^{\mathcal{A}} & \text{for each } a \in V_{\mathsf{addr}} \\
c^{\mathcal{B}} & = & c^{\mathcal{A}} & \text{for each } c \in V_{\mathsf{cell}} \\
t^{\mathcal{B}} & = & t^{\mathcal{A}} & \text{for each } t \in V_{\mathsf{thid}} \\
m^{\mathcal{B}}(v) & = & \left(m^{\mathcal{A}}(v).data^{\mathcal{A}}, first(m^{\mathcal{A}}, v, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{A}}(v).lockid^{\mathcal{A}}\right) & \text{for each } m \in V_{\mathsf{mem}},\ v \in \mathcal{B}_{\mathsf{addr}} \\
s^{\mathcal{B}} & = & s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} & \text{for each } s \in V_{\mathsf{set}} \\
g^{\mathcal{B}} & = & g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} & \text{for each } g \in V_{\mathsf{settid}} \\
p^{\mathcal{B}} & = & compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) & \text{for each } p \in V_{\mathsf{path}}
\end{array}
$$

Clearly, by construction $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{thid}}$ and $\mathcal{B}_{\mathsf{elem}}$ satisfy the given cardinality constraints.

The proof that $\mathcal{B}$ satisfies all TLL-literals in Γ is not shown here since, as we said, many TLL3-literals are shared with TLL. The proof for the shared literals can be found in [RZ06b]. Here we just limit ourselves to the new literals defined in TLL3. Hence, we consider the following cases:

Literals of the form $t_1 \neq t_2$: Immediate.

Literals of the form $c = mkcell(e, a, t)$: We know that

$$
\begin{array}{rcl}
c^{\mathcal{B}} & = & c^{\mathcal{A}} \\
& = & mkcell^{\mathcal{A}}(e, a, t) \\
& = & \langle e^{\mathcal{A}}, a^{\mathcal{A}}, t^{\mathcal{A}} \rangle \\
& = & \langle e^{\mathcal{B}}, a^{\mathcal{B}}, t^{\mathcal{B}} \rangle \\
& = & mkcell^{\mathcal{B}}(e, a, t)
\end{array}
$$

Literals of the form $c = rd(m, a)$: In this case we have that

$$
\begin{array}{rcl}
\left[rd(m, a)\right]^{\mathcal{B}} & = & m^{\mathcal{B}}(a^{\mathcal{B}}) \\
& = & m^{\mathcal{B}}(a^{\mathcal{A}}) \\
& = & \left(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, first(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}\right)
\end{array}
$$

$$
\begin{aligned}
&= \left(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}, m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}\right) && \text{(Lemma 4.2(a))}\\
&= m^{\mathcal{A}}(a^{\mathcal{A}})\\
&= c^{\mathcal{A}}\\
&= c^{\mathcal{B}}
\end{aligned}
$$

Literals of the form $m = upd(\tilde{m}, a, c)$**:** In this particular case we want to prove that $m^{\mathcal{B}}(a^{\mathcal{B}}) = c^{\mathcal{B}}$ while for any other $v \in \mathcal{B}_{\mathsf{addr}}$, such that $v \neq a^{\mathcal{B}}$, $m^{\mathcal{B}}(v) = \tilde{m}^{\mathcal{B}}(v)$. Since $m^{\mathcal{A}}(a^{\mathcal{A}}) = c^{\mathcal{A}}$, then we have that $c^{\mathcal{B}} = m^{\mathcal{B}}(a^{\mathcal{B}})$. Let now $v \neq a^{\mathcal{B}}$. Then,

$$
\begin{aligned}
m^{\mathcal{B}}(v) &= \left(m^{\mathcal{A}}(v).data^{\mathcal{A}}, first(m^{\mathcal{A}}, v, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{A}}(v).lockid^{\mathcal{A}}\right)\\
&= \left(\tilde{m}^{\mathcal{A}}(v).data^{\mathcal{A}}, first(\tilde{m}^{\mathcal{A}}, v, \mathcal{B}_{\mathsf{addr}}), \tilde{m}^{\mathcal{A}}(v).lockid^{\mathcal{A}}\right) && \text{(Lemma 4.2(b))}\\
&= \tilde{m}^{\mathcal{B}}(v)
\end{aligned}
$$

Literals of the form $a = fstlock(m, p)$**:** First consider the case $p = \epsilon$. Hence $fstlock^{\mathcal{A}}(m^{\mathcal{A}}, \epsilon^{\mathcal{A}}) = null^{\mathcal{A}}$. At the same time, we know that $\epsilon^{\mathcal{B}} = compress(\epsilon^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$ and so $fstlock^{\mathcal{B}}(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}) = null^{\mathcal{B}}$. Let's now consider the case in which $p = [a_1, \ldots, a_n]$. There are two possible scenarios to consider.

- If for all $1 \leq k \leq n$, $m^{\mathcal{A}}(a_k^{\mathcal{A}}).lockid^{\mathcal{A}} = \oslash$, then we have that

  $$fstlock^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}) = null^{\mathcal{A}}$$

  Notice that function *compress* returns a subset of the path it receives with the property that all addresses in the returned path belong to the received path. Therefore, if we have $[\tilde{a}_1, \ldots, \tilde{a}_m] = p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$, we can deduce that $\{\tilde{a}_1, \ldots, \tilde{a}_m\} \subseteq \mathcal{B}_{\mathsf{addr}}$ and hence for all $1 \leq j \leq m$, $m^{\mathcal{B}}(\tilde{a}_j).lockid^{\mathcal{B}} = \oslash$. Then, we conclude that $fstlock^{\mathcal{B}}(m^{\mathcal{B}}, p^{\mathcal{B}}) = null^{\mathcal{B}}$.

- If there is a $1 \leq k \leq n$ s.t., for all $1 \leq j < k$, $m^{\mathcal{A}}(a_j^{\mathcal{A}}).lockid^{\mathcal{A}} = \oslash$ and $m^{\mathcal{A}}(a_k^{\mathcal{A}}).lockid^{\mathcal{A}} \neq \oslash$ then since by the construction of model $\mathcal{B}$, we have that $a^{\mathcal{B}} = a^{\mathcal{A}}$. Therefore, we can say that $a^{\mathcal{B}} = a^{\mathcal{A}} = x \in \mathcal{B}_{\mathsf{addr}}$. It then remains to verify whether

  $$x = fstlock^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}) \quad \rightarrow \quad x = fstlock^{\mathcal{B}}(m^{\mathcal{B}}, compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}))$$

  By definition of *fstlock* we have that $x = a_k^{\mathcal{A}}$ and by the construction of set $\mathcal{B}_{\mathsf{addr}}$, we know that $a_k^{\mathcal{A}} \in \mathcal{B}_{\mathsf{addr}}$. Let $[\tilde{a}_1, \ldots, \tilde{a}_i, \ldots, \tilde{a}_m] = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$ such that $\tilde{a}_i = a_k^{\mathcal{A}}$. It is clear that $\tilde{a}_j \in \mathcal{B}_{\mathsf{addr}}$ for all $1 \leq j \leq m$. Then, as *compress* preserves the order and for all $1 \leq j < k$, $m^{\mathcal{A}}(a_j^{\mathcal{A}}).lockid^{\mathcal{A}} = \oslash$, we have that for all $1 \leq j < i$, $m^{\mathcal{B}}(\tilde{a}_j).lockid^{\mathcal{B}} = \oslash$. Besides $m^{\mathcal{B}}(\tilde{a}_i).lockid^{\mathcal{B}} \neq \oslash$. Then:

  $$
  \begin{aligned}
  fstlock^{\mathcal{B}}(m^{\mathcal{B}}, compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) &= fstlock^{\mathcal{B}}(m^{\mathcal{B}}, [\tilde{a}_1, \ldots, \tilde{a}_m])\\
  &= \tilde{a}_i\\
  &= a^{\mathcal{A}}\\
  &= x
  \end{aligned}
  $$

  □

4.2 A Combination-based Decision Procedure for TLL3

Lemma 4.3 justifies a brute force method to automatically check TLL3 satisfiability of normalized TLL3-literals. However, such a method is not efficient in practice. To find a more efficient decision procedure we decompose TLL3 into a combination of theories, and apply a many-sorted variant of the Nelson-Oppen combination method [TZ04].

In [NO79], Nelson and Oppen show how a decision procedure for a theory built by many first-order theories can be derived as a combination of the existing decision procedures for each of these theories. The main idea is to combine the decision procedures by means of equality sharing, guessing an arrangement over the set of shared variables. This arrangement is used to build equalities and disequalities between variables, to constrain simultaneously the inputs of decision procedures for each of the component theories.

This method requires the theories to fulfill two conditions. First, each theory must have a decision procedure. Second, all involved theories must be stable infinite and share sorts only.

Definition 4.3 (stable-infiniteness).
A Σ-theory T is stably infinite if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ whose domain is infinite. †

As we said, TLL [RZ06a] is a union of theories. All theories involved in TLL are stably-infinite, so the only missing theories are T_{settid} and the one defining *fstlock*. Clearly T_{settid} is stably infinite (following a similar reasoning as for T_{set}). This leaves us only with the problem of finding a decision procedure for Σ_{reach} and a way to define the functions and predicates in Σ_{bridge}. We begin by defining the theory T_{Base3} as follows:

$$T_{\mathsf{Base3}} = T_{\mathsf{addr}} \cup T_{\mathsf{elem}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{path}} \cup T_{\mathsf{set}} \cup T_{\mathsf{settid}} \cup T_{\mathsf{thid}}$$

where T_{path} extends the theory of finite sequences of addresses with the auxiliary functions and predicates depicted in Fig. 4.3. Moreover, we say that Σ_{Base3} is the signature obtained by the union of all the signatures for each theory that is part of T_{Base3}. The theory of finite sequences of addresses is defined by $T_{\mathsf{fseq}} = \left(\Sigma_{\mathsf{fseq}}, \mathsf{TGen}\right)$, where:

$$\begin{array}{rcl}
\Sigma_{\mathsf{fseq}} & = & \Big(\{\ \mathsf{addr}, \mathsf{fseq} \qquad\qquad\qquad\qquad\ \}, \\
& & \ \{\ \mathit{nil} \ : \ \mathsf{fseq}, \\
& & \quad \mathit{cons} \ : \ \mathsf{addr} \times \mathsf{fseq} \to \mathsf{fseq}, \\
& & \quad \mathit{hd} \ : \ \mathsf{fseq} \to \mathsf{addr}, \\
& & \quad \mathit{tl} \ : \ \mathsf{fseq} \to \mathsf{fseq} \qquad\ \}, \\
& & \{\qquad\qquad\qquad\qquad\qquad\ \}\ \Big)
\end{array}$$

and TGen is the class of multi-sorted term-generated structures that satisfy the axioms of T_{fseq}. These axioms are the standard for a theory of lists, such as distinctness, uniqueness and generation of sequences using the constructors *cons* and *nil*, as well as acyclicity of sequences (see, for example [BM07]). Let *PATH* be the set of axioms of T_{fseq} including all in Fig. 4.3. Using these definitions, we can formally define $T_{\mathsf{path}} = \left(\Sigma_{\mathsf{path}}, \mathsf{ETGen}\right)$ where ETGen is $\left\{\mathcal{A}^{\Sigma_{\mathsf{path}}} \,|\, \mathcal{A}^{\Sigma_{\mathsf{path}}} \models \mathit{PATH} \text{ and } \mathcal{A}^{\Sigma_{\mathsf{fseq}}} \in \mathsf{TGen}\right\}$.

Next, we extend T_{Base3} defining the missing functions and predicates from T_{reach} and Σ_{bridge}. For *addr2set* and *reach* the definition is immediate from *fseq2set* and $\mathit{isreachp}_{\mathsf{K}}$ respectively. For the other

functions and predicates we can use the following definitions:

$$
\begin{array}{rcl}
nil & = & \epsilon \\
cons(a, nil) & = & [a] \\
ispath(p) \rightarrow fseq2set(p) & = & path2set(p) \\
\left(\begin{array}{ll} ispath(p_1) \wedge ispath(p_2) & \wedge \\ fseq2set(p_1) \cap fseq2set(p_2) = \emptyset & \wedge \\ app(p_1, p_2) = p_3 & \end{array}\right) & \leftrightarrow & append(p_1, p_2, p_3) \\
isreachp(m, a_1, a_2, p) & \rightarrow & getp(m, a_1, a_2) = p \\
\neg isreachp(m, a_1, a_2, p) & \rightarrow & getp(m, a_1, a_2) = nil \\
ispath\,(p) \wedge fstmark\,(m, p, i) & \leftrightarrow & fstlock\,(m, p) = i
\end{array}
$$

Let *GAP* be the set of axioms that define ϵ, [_], *append*, *reach*, *path2set*, *addr2set* and *getp*. We define $\widehat{\mathsf{TLL3}} = (\Sigma_{\widehat{\mathsf{TLL3}}}, \widehat{\mathsf{ETGen}})$ where $\Sigma_{\widehat{\mathsf{TLL3}}}$ is $\Sigma_{\mathsf{Base3}} \cup \{getp, append, path2set, fstlock\}$ and $\widehat{\mathsf{ETGen}}$ is $\left\{\mathcal{A}^{\Sigma_{\widehat{\mathsf{TLL3}}}} | \mathcal{A}^{\Sigma_{\widehat{\mathsf{TLL3}}}} \models GAP \text{ and } \mathcal{A}^{\Sigma_{\mathsf{path}}} \in \mathsf{ETGen}\right\}$.

Using the definitions of *GAP* it is easy to prove that if Γ is a set of normalized TLL3-literals, then Γ is TLL3-satisfiable iff Γ is $\widehat{\mathsf{TLL3}}$-satisfiable. It is enough to carry out an analysis by cases. For all literals belonging to both theories, this verification is straightforward. On the other hand, for literals of the form involving *path2set*, *append* or *getp* for instance, we just need to apply the definitions introduced by *GAP* and then apply the definitions of *PATH*.

This way, $\widehat{\mathsf{TLL3}}$ can be used in place of TLL3 for satisfiability checking. We reduce $\widehat{\mathsf{TLL3}}$ into T_{Base3} in two steps. First we do the unfolding of the definition of auxiliary functions defined in *PATH* and *GAP*,

$app : \mathsf{fseq} \times \mathsf{fseq} \rightarrow \mathsf{fseq}$
$app(nil, l) = l$
$app(cons(a, l), l') = cons(a, app(l, l'))$

$fseq2set : \mathsf{fseq} \rightarrow \mathsf{set}$
$fseq2set(nil) = \emptyset$
$fseq2set(cons(a, l)) = \{a\} \cup fseq2set(l)$

$ispath : \mathsf{fseq}$
$ispath(nil)$
$ispath(cons(a, nil))$
$\{a\} \not\subseteq fseq2set(l) \wedge ispath(l) \rightarrow ispath(cons(a, l))$

$last : \mathsf{fseq} \rightarrow \mathsf{addr}$
$last(cons(a, nil)) = a$
$l \neq nil \rightarrow last(cons(a, l)) = last(l)$

$isreach : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr}$
$isreach(m, a, a)$
$m[a].next = a' \wedge isreach(m, a', b) \rightarrow isreach(m, a, b)$

$isreachp : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{fseq}$
$isreachp(m, a, a, nil)$
$m[a].next = a' \wedge isreachp(m, a', b, p) \rightarrow isreachp(m, a, b, cons(a, p))$

$fstmark : \mathsf{mem} \times \mathsf{fseq} \times \mathsf{addr}$
$fstmark(m, nil, null)$
$p \neq nil \wedge p = cons(j, q) \wedge m[j].lockid \neq \oslash \rightarrow fstmark(m, p, j)$
$p \neq nil \wedge p = cons(j, q) \wedge m[j].lockid = \oslash \wedge fstmark(m, q, i) \rightarrow fstmark(m, p, i)$

Figure 4.3: Functions, predicates and axioms of T_{path}

$$List : \mathsf{mem} \times \mathsf{addr} \times \mathsf{set}$$
$$List(h, a, r) \quad \leftrightarrow \quad null \in addr2set(h, a) \wedge r = path2set(getp(h, a, null))$$

$$f_a : \mathsf{mem} \times \mathsf{addr} \rightarrow \mathsf{path}$$
$$f_a(h, n) = \begin{cases} \epsilon & \text{if } n = null \\ getp(h, h[n].next, null) & \text{if } n \neq null \end{cases}$$

$$f_b : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{path}$$
$$\begin{aligned} f_b(h, n, m, p) \quad \leftrightarrow \quad & (n = null \wedge p = \epsilon) \quad \vee \\ & \Big(n \neq null \wedge reach(h, n, m, \tilde{p}) \quad \wedge \quad m = null \rightarrow append(p, null, \tilde{p}) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad m \neq null \rightarrow append(p, [m], \tilde{p})\Big) \end{aligned}$$

$$LastMarked : \mathsf{mem} \times \mathsf{path} \rightarrow \mathsf{addr}$$
$$LastMarked(m, p) = fstlock(m, rev(p))$$

$$NoMarks : \mathsf{mem} \times \mathsf{path}$$
$$NoMarks(m, p) \leftrightarrow fstlock(m, p) = null$$

$$SomeMark : \mathsf{mem} \times \mathsf{path}$$
$$SomeMark(m, p) \leftrightarrow fstlock(m, p) \neq null$$

Figure 4.4: Auxiliary functions to reason about concurrent lists

getting rid of the extra functions, and obtaining a formula in T_{Base3}. Second, as we have proved that TSL_K enjoys finite model property then it is always possible to enumerate the finitely many ground terms. This means that we can always find a model with a finite number of elements that preserves satisfiability of a conjunction of TSL_K literals. Hence, by substituting all possible ground terms in the set of normalized literals and symbolically executing the definitions of the symbols in *PATH* it is possible to reduce the $\widehat{\mathsf{TLL3}}$-satisfiability problem of normalized literals to the T_{Base3}-satisfiability of quantifier free formulas, setting a bound for the unfolding of recursive formulas. This way it is possible to obtain a decision procedure for reachability, based on the finite model property of the theory.

All theories involved in T_{Base3} share only sorts symbols, are stably-infinite and for all of them there is a decision procedure. Hence, the multi-sorted Nelson-Oppen combination method can be applied, obtaining a decision procedure for TLL3.

We now define some auxiliary functions and predicates using TLL3, that aid in the reasoning about concurrent linked-lists (see Fig. 4.4). For example, predicate $List(h, a, r)$ expresses that in heap h, starting from address a there is sequence of cells all of which form region r. Function f_a, given a heap h and an address n, returns the path that goes from the address pointed by the cell stored in n up to the end of the list (i.e., a *null* pointer is found). Similarly, predicate f_b holds when p is the path that connects the node stored at address n with the one at address m. Notice that m is not considered as part of the path. Function $LastMarked(h, p)$, on the other hand, returns the address of the last locked node in path p on memory h. All these functions can be used in verification conditions. Then, using the equivalences in Fig. 4.4 the predicates are removed, generating a pure $\widehat{\mathsf{TLL3}}$ formula whose satisfiability can be checked with the procedure described above.

4.3 Verifying Some Properties Over Concurrent Lists

In this section we sketch the proofs for some properties over lock-coupling singly-linked lists. The reader will notice that all literals involved in the formulas described from this point on belong to TLL3. This way, the decision procedure presented in this chapter can be used to automatically verify them all.

Firstly, we prove that the thread owning the locks closest to the tail of the list, eventually terminates. We have already introduced this property on Example 3.1. Finally, we present the idea behind the verification of the property describing that no thread can overtake other thread that already owns some lock in the list.

4.3.1 Termination of Concurrent Lock-Coupling Lists

In this section we show the proof of a simple liveness property of concurrent lock-coupling lists: termination of the leading thread. To aid in the verification of this property we annotate the code with ghost fields and ghost updates. We use boxes to represent the annotations introduced in the code. We enrich *List* objects introduced in Section 3.1 with a ghost field r of type region that keeps track of all the nodes in the list:

class *List* {
  Node list;
  rgn *r*;
}

class *Node* {
  Value val;
  Node next;
  Lock lock;
}

The algorithms for INSERT and REMOVE are shown as Algorithms 4.1 and 4.2 respectively.

The predicate $c.lockid = \oslash$ denotes that the lock of list node c is not taken. The predicate $c.lockid = k$ establishes that the lock at list node c is owned by thread k. As shown, the code for INSERT and REMOVE is extended with ghost updates to maintain r.

T_k denotes thread k. We want to prove that if a thread has acquired a lock at node n and no other thread holds a lock ahead of n, then thread k eventually terminates. The predicate $at\_\textsc{Insert}_n^{[k]}$ means that thread k is executing line n of program INSERT. Similarly, $at\_\textsc{Insert}_{n_1,\ldots,n_m}^{[k]}$ is a short for thread k is running some of the lines $n_1,\ldots,n_m$ of program INSERT. We use $n..m$ to denote all correlative lines from n to m. The instance of a local variable v in thread k is represented by $v^{[k]}$. We define *DisjList* as an extension of *List* enriching it with the property that new nodes created during insertion are all disjoint from each other, including all nodes that are already part of the list:

$$\begin{aligned} \mathit{DisjList}(h,a,r) \;\hat{=}\; & \mathit{List}(h,a,r) & \wedge \\ & \forall j : T_{ID} \,.\, at\_\textsc{Insert}_{5,6}^{[j]} \rightarrow \left\langle aux^{[j]} \right\rangle \# r & \wedge \\ & \forall i,j : T_{ID}. i \neq j \wedge at\_\textsc{Insert}_{5,6}^{[i]} \wedge at\_\textsc{Insert}_{5,6}^{[j]} \rightarrow \left\langle aux^{[i]} \right\rangle \# \left\langle aux^{[j]} \right\rangle \# r \end{aligned}$$

We now define the following auxiliary predicate:

$$\begin{aligned} \mathit{IsLast}(k) \;\hat{=}\; & \mathit{DisjList}(h, l.list, l.r) & \wedge \\ & \mathit{SomeMark}\big(h, \mathit{getp}(h, l.list, null)\big) & \wedge \\ & \mathit{LastMarked}\big(h, \mathit{getp}(h, l.list, null)\big) = a & \wedge \\ & h[a].lockid = k \end{aligned}$$

Algorithm 4.1 Insertion for concurrent lock-coupling lists extended with ghost code

```
1:  procedure INSERT( Value e)
2:      prev, curr := locate(e)
3:      if curr.val ≠ e then
4:          aux := new Node(e)
5:          aux.next := curr
6:          prev.next := aux        ——— l.r := l.r ∪ {⟨aux⟩}
7:          result := true
8:      else
9:          result := false
10:     end if
11:     prev.unlock()
12:     curr.unlock()
13:     return result
14: end procedure
```

Algorithm 4.2 Remove for concurrent lock-coupling lists extended with ghost code

```
1:  procedure REMOVE(Value e)
2:      prev, curr := locate(e)
3:      if curr.val = e then
4:          aux := curr.next
5:          prev.next := aux        ——— l.r := l.r − {⟨aux⟩}
6:          result := true
7:      else
8:          result := false
9:      end if
10:     prev.unlock()
11:     curr.unlock()
12:     return result
13: end procedure
```

Notice how the ghost variable r of type region is used in the *List* predicate to assert that new nodes do not already belong to the list. The formula $IsLast(k)$ identifies whether T_k is the thread owning the last lock in the list (i.e., the closest node towards the end of the list). Using these predicates we define the parametrized temporal formula we want to verify as:

$$\psi(k) \triangleq \Box\left(at\_\textsc{Locate}^{[k]}_{4..11} \wedge IsLast(k) \rightarrow IsLast(k)\,\mathcal{U}\; at\_\textsc{Locate}^{[k]}_{12}\right)$$

This temporal formula states that if thread k is running Locate and it owns the last locked node in the list, then thread T_k will still own the last locked node until T_k reaches the last line of Locate. Reachability of the last line of Locate implies termination of the invocation to the concurrent datatype because Locate is the only program containing potentially blocking operations.

We proceed with the construction of a verification diagram that proves that the parallel execution of all threads guarantees the satisfaction of formula $\psi(k)$. Given N, we build the transition system $\mathcal{S}[N]$, in which threads $T_1, \ldots, T_N$ run in parallel the program MGC and show that $\mathcal{S}[N] \vDash \psi(k)$. The verification diagram is depicted in Fig. 4.5.

We use $\tau^{[k]}_{p_n}$ to denote the transition taken by thread k on program p at line n, and $\tau^{[N]}_{p_n}$ to describe he set $\tau^{[j]}_{p_n}$, for all $j \in \{1..N\}$.

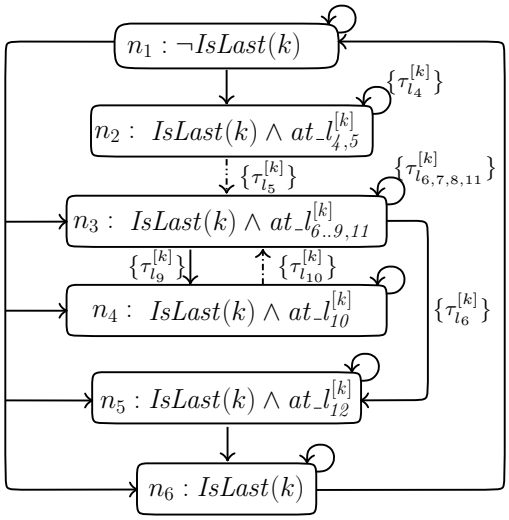


Figure 4.5: Verification diagram Ψ for $\|_{j<N}\, T_j \vDash \psi(k)$

Dashed arrows in the diagram denote transitions that strictly decrement the ranking function δ. Formally, the verification diagram is defined by:

- $N_0 = \{n_1\}$
- $\mathcal{F} = \{(P, R)\}$, with $P = \{(n_3, n_4), (n_3, n_5), (n_5, n_6), (n_6, n_1)\} \cup \{(n_1, n_j) | j \in 2..6\} \cup \{(n_j, n_j) | j \in 1..6\}$ and $R = \emptyset$
- $\delta(n, s) = \begin{cases} \left\{ a \mid a \in dom(h) \right\} & n = n_1, n_2 \\ path2set\left(f_a(h, LastMarked(h, getp(h, prev^{[k]}, null)))\right) & \text{otherwise} \end{cases}$
- $f(n) = \begin{cases} \emptyset & \text{if } n = n_1, n_6 \\ at\_\text{Locate}^{[k]}_{4,5} & \text{if } n = n_2 \\ at\_\text{Locate}^{[k]}_{6..9,11} & \text{if } n = n_3 \\ at\_\text{Locate}^{[k]}_{10} & \text{if } n = n_4 \\ at\_\text{Locate}^{[k]}_{12} & \text{if } n = n_5 \end{cases}$

We can now describe the verification conditions:

initialization Trivial, since in the initial state *l.list* forms an empty list, and consequently $\neg IsLast(k)$.

consecution We will show, for illustration purposes, transition $\tau^{[j]}_{\text{Locate}_{10}}$ on node n_2 with $j \neq k$. The verification condition is:

$$\left(\underbrace{\overbrace{IsLast(k) \wedge}^{\text{TLL3}} \overbrace{j \neq k}^{T_{\text{thid}}} \wedge \; at\_l^{[k]}_{4,5} \wedge at\_l^{[j]}_{10} \wedge \; curr^{[j]}.lockid = \oslash}_{\text{TLL3}} \right) \wedge \overbrace{curr^{[j]}.lock(j)}^{\text{TLL3}} \quad \rightarrow \quad \left(\underbrace{\overbrace{IsLast(k') \wedge at'\_l^{[k']}_{4,5} \wedge at'\_l^{[j']}_{11} \wedge pres(V - curr^{[j]}) \wedge}^{\text{TLL3}} \; curr'^{[j']}.lockid = j'}_{\text{TLL3}} \right)$$

where *pres* is the predicate denoting variable preservation. Note that all fragments of such verification condition belong to theories for which we have already defined a decision procedure, including propositional logic for the (finite) locations of the program counters.

acceptance The ranking function δ maps, at a given state, the set of list nodes accessible from the last node with an owned lock. This set remains identical for all transitions except $\tau^{[k]}_{l_5}$ and $\tau^{[k]}_{l_{10}}$, for which the set decrements (in the inclusion order on sets). The decision procedure presented in this chapter proves this automatically (using $\subset$ operation and equality over sets of addresses).

fairness Only two conditions must be verified. First, all transitions labeling an edge are enabled since the only potentially blocking operation is $\tau^{[k]}_{l_{10}}$ and $IsLast(k)$ implies that $\tau^{[k]}_{l_{10}}$ is enabled. Second, for all nodes and labeled edges, starting from a state that satisfies the predicate of the incoming node satisfies the predicate of the outgoing node via taking the transition. Sequential progress of thread k is guaranteed by fairness, since all idling transitions for thread k are in fact a diagram idiom to represent the expansion of such nodes to a sequence of nodes with a single program position on each node.

satisfaction $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\psi(k))$ is automatically checkable via a finite LTL model-checking problem. In this case, instead of proving that all accepting paths in the diagram are contained into the set of sequences satisfying formula $\psi(k)$, we show that the intersection with the negation of the formula is empty. We use $\langle v_1, v_2, v_3 \rangle$ to represent $\langle at\_l^{[k]}_{4..11}, at\_l^{[k]}_{12}, IsLast(k) \rangle$. In fact $IsLast(k)$ should be decomposed into all its atomic subformulas. However, for the sake of simplicity we assume that an

assignment to *IsLast*(*k*) represents all necessarily assignments to its atomic subformulas in order to make *IsLast*(*k*) predicate true. Then, since

$$\neg\psi(k) = \Diamond\left(at\_l_{4..11}^{[k]} \wedge IsLast(k) \wedge \left(\neg at\_l_{12}^{[k]}\ \mathcal{W}\ \neg IsLast(k)\right)\right)$$

we have that:

$$\mathcal{L}^P(\neg\psi(k)) = \langle -,-,-\rangle^*\langle \mathrm{T},\mathrm{F},\mathrm{T}\rangle\left(\langle -,\mathrm{F},-\rangle^\omega \cup \langle -,\mathrm{F},-\rangle^*\langle -,-,\mathrm{F}\rangle\langle -,-,-\rangle^\omega\right)$$

while for our verification diagram we have

$$\left(\langle -,-,\mathrm{F}\rangle^+\langle \mathrm{T},\mathrm{F},\mathrm{T}\rangle^*\langle \mathrm{F},\mathrm{T},\mathrm{T}\rangle^+\langle -,-,\mathrm{T}\rangle^+\right)^\omega$$

Imagine we consider the sequence $\langle -,-,-\rangle^*\langle \mathrm{T},\mathrm{F},\mathrm{T}\rangle\langle -,\mathrm{F},-\rangle^\omega$. Then, $\langle \mathrm{T},\mathrm{F},\mathrm{T}\rangle$ should correspond to $\langle \mathrm{T},\mathrm{F},\mathrm{T}\rangle^*$. Then, it is impossible to match $\langle -,\mathrm{F},-\rangle^\omega$ with any possible pattern in the accepting paths of the diagram. On the other hand, if we consider the sequence $\langle -,\mathrm{F},-\rangle^*\langle -,-,\mathrm{F}\rangle\langle -,-,-\rangle^\omega$, notice that there is no possible way to match $\langle -,-,\mathrm{F}\rangle$ with the accepting paths of the diagram. This way we have shown that both languages are disjoint.

4.3.2 No Thread Overtakes

In this section we sketch the proof that it is not possible in the presented implementation that, once a thread has acquired a lock, being overtaken by other thread. To carry out the proof, we first extend the code for concurrent lock-coupling singly-linked lists with two ghost variables: a global ghost variable *ticket* and a local ghost variable *myTicket*. Then, we modify Locate to consider the new ghost variables. The extended version of the Locate program with the corresponding modification is depicted in Algorithm 4.3.

Algorithm 4.3 Locate program for preventing overtake

```
1:  procedure LOCATE(Value e)
2:      prev := Head
3:      prev.lock()                          ——— myTicket := ticket
4:      curr := prev.next                            ticket := ticket + 1
5:      curr.lock()
6:      while curr.val < e do
7:          prev.unlock()
8:          prev := curr
9:          curr := curr.next
10:         curr.lock()
11:     end while
12:     return (prev, curr)
13: end procedure
```

The idea is that every time a thread gets the first lock in the list, it also gets a ticket number. Tickets are delivered by the data structure and they are strictly increasingly. This policy guarantees that no thread gets a duplicated ticket and threads are ordered according to the order in which they acquired their first lock. We want to assure that there is no overtake in the system. This means that all threads remain ordered as they own a ticket. To express the formula that describes this condition we first declare the *ahead* predicate defined over thread identifiers t_1 and t_2 and a memory configuration m:

$$ahead(t_1,t_2) \quad \hat{=} \quad haveLocks(t_1,t_2) \rightarrow \left(\begin{array}{ll} fstlock(m, prev^{[t_1]}) \neq null & \wedge \\ fstlock(m, prev^{[t_2]}) \neq null & \wedge \\ fstlock(m, prev^{[t_2]}) \in addr2set(m, curr^{[t_1]}) & \end{array}\right)$$

where

$$haveLocks(t_1, t_2) \quad = \quad at\_\textsc{Locate}^{[t_1,t_2]}_{4..12} \vee at\_\textsc{Insert}^{[t_1,t_2]}_{3..12} \vee at\_\textsc{Remove}^{[t_1,t_2]}_{3..11} \vee at\_\textsc{Search}^{[t_1,t_2]}_{3..9}$$

In the formula above, we use $at\_p_n^{[t_1,t_2]}$ to denote $at\_p_n^{[t_1]} \wedge at\_p_n^{[t_2]}$. Roughly speaking, predicate *ahead* holds when both threads have at least one locked node and the leftmost node locked by thread T_2 (that is, the nearest to the head of the list) is between the position of t_1 and the tail of the list.

We can now use the *ahead* predicate to define the condition we want to verify. We use φ to represent such formula:

$$\varphi \quad \hat{=} \quad \Big(myTicket^{[t_1]} > myTicket^{[t_2]}\Big) \quad \rightarrow \quad \Box\Big(ahead(t_1, t_2) \rightarrow \bigcirc\Big(ahead(t_1, t_2)\Big)\Big)$$

Once more, notice how ghost variables *myTicket* and *ticket* are used and are part of formula φ. Now, φ is a system invariant; we just need to verify that:

- φ is satisfied by the system's initial condition, and
- all transitions preserve φ.

It is easy to see that the initial condition satisfies φ. At such moment, no ticket number has been assigned nor any thread has acquired any lock. Then, formula φ is trivially satisfied.

Now we analyze that every transition preserves φ. In fact, here we do not analyze all possible transitions, but we just focus on the possible offending ones. That is, transitions that get or release a lock, or that modifies the position of the *curr* pointer. Notice that all these transitions can potentially modify the validity of φ. All other transitions are not relevant as the modification they introduced cannot modify the value of φ at all or they are simply trivially valid because of propositional reasoning on the program location. Now, the transitions we analyze are:

- When a threads acquires its first lock. For an arbitrary thread t, this transition corresponds to $at\_\textsc{Locate}^{[t]}_3$. This transition clearly satisfies φ' because the value of *ticket* is strictly increasing. This guarantees that for all other thread s with locks in the list, $myTicket^{[t]} > myTicket^{[s]}$ and $ahead(t, s)$ hold, since $fstlock(m, s) \in addr2set(m, prev^{[t]})$.
- When a thread gets a new lock. This modification is accomplished by the transitions $at\_\textsc{Locate}^{[t]}_5$ and $at\_\textsc{Locate}^{[t]}_{10}$. If some of these transitions are taken by thread t_2 then φ is clearly preserved. If the transition is taken by t_1, the ticket numbering and order is preserved. Besides, t_1 acquires its new lock over a node where no other thread has a lock. Hence, once more we conclude that $fstlock(m, t_2) \in addr2set(m, prev^{[t_1]})$.
- When a thread releases its last lock. This happens when transitions $at\_\textsc{Insert}^{[t]}_{12}$, $at\_\textsc{Remove}^{[t]}_{11}$ or $at\_\textsc{Search}^{[t]}_9$ are taken. Notice that in this case one of the thread is moving outside of the range set by the *haveLocks* predicate. Hence, $\bigcirc ahead(t_1, t_2)$ trivially holds and therefore φ does.
- When the position of a thread advances through the nodes in the lists. This progress corresponds to transitions $at\_\textsc{Locate}^{[t]}_4$ and $at\_\textsc{Locate}^{[t]}_9$. In this case ticket numbers are not modified nor any lock is acquired or removed. Hence, φ clearly holds.

5

TSL_K: A Decision Procedure for Concurrent Skiplists

In this chapter, we present a theory of skiplists with a decidable satisfiability problem, and show its applications to the verification of concurrent skiplist implementations. As already described in Section 3.2, a skiplist is a data structure used to implement sets by maintaining several ordered singly-linked lists in memory, with a performance comparable to balanced binary trees. We define a theory capable of expressing the memory layout of a skiplist and show a decision procedure for the satisfiability problem of this theory. Concurrent lock-coupling skiplists introduced in Chapter 3, where every node contains a lock at each possible level, reduce granularity of mutual exclusion sections.

We now proceed with the description of the theory TSL_K. This theory is briefly described in [SS11]. TSL_K is a decidable theory capable of reasoning about list reachability, locks, ordered lists, and sublists of ordered lists. First, we show that TSL_K enjoys a finite model property and thus it is decidable. Finally, we show how to reduce the satisfiability problem of quantifier-free TSL_K formulas to a combination of theories for which a many-sorted version of Nelson-Oppen can be applied.

We build a decision procedure to reason about skiplist of height K combining different theories, aiming to represent pointer data structures with a skiplist layout, masked regions and locks. We extend the Theory of Concurrent Linked Lists (TLL3) presented in Chapter 4, a decidable theory that includes reachability of concurrent list-like structures, in the following way:

- each node is equipped with a *key* field, used to reason about node's order.
- the reasoning about single level lists is extended to all the K levels.
- we extend the theory of regions with masked regions.
- lists are extended to ordered lists and sub-paths of ordered lists.

Formally, the theory of skiplists of height K is defined as $\mathsf{TSL}_\mathsf{K} = (\Sigma_{\mathsf{TSL}_\mathsf{K}}, \mathbf{TSLK})$, where

$$\Sigma_{\mathsf{TSL}_\mathsf{K}} = \Sigma_{\mathsf{level}_\mathsf{K}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{reach}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{settid}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{aarr}} \cup \Sigma_{\mathsf{larr}} \cup \Sigma_{\mathsf{bridge}}$$

Then, the signature of TSL_K is shown in Fig. 5.1 and 5.2. **TSLK** is the class of $\Sigma_{\mathsf{TSL}_\mathsf{K}}$-structures satisfying the conditions depicted in Fig. 5.3 and 5.4. The symbols of Σ_{set} and Σ_{settid} follow their standard

Signature:	$\Sigma_{\mathsf{level_K}}$		
Sorts:	$\mathsf{level_K}$		
Functions:	0,1, ..., K-1	:	$\mathsf{level_K}$
Predicates:	$<$	:	$\mathsf{level_K} \times \mathsf{level_K}$
Signature:	Σ_{ord}		
Sorts:	ord		
Functions:	$-\infty, +\infty$	:	ord
Predicates:	$\preceq$	:	ord $\times$ ord
Signature:	Σ_{thid}		
Sorts:	thid		
Functions:	$\oslash$	:	thid
Signature:	Σ_{cell}		
Sorts:	cell, elem, ord, $\mathsf{level_K}$, addr, thid		
Functions:	*error*	:	cell
	mkcell	:	elem $\times$ ord $\times$ $\mathsf{addr^K} \times \mathsf{thid^K} \rightarrow$ cell
	_.*data*	:	cell $\rightarrow$ elem
	_.*key*	:	cell $\rightarrow$ ord
	.*next* [ ]	:	cell $\times$ $\mathsf{level_K} \rightarrow$ addr
	.*lockid* [ ]	:	cell $\times$ $\mathsf{level_K} \rightarrow$ thid
	.*lock* [ ]	:	cell $\times$ $\mathsf{level_K} \rightarrow$ thid $\rightarrow$ cell
	.*unlock* [ ]	:	cell $\times$ $\mathsf{level_K} \rightarrow$ cell
Signature:	Σ_{mem}		
Sorts:	mem, addr, cell		
Functions:	*null*	:	addr
	_[_]	:	mem $\times$ addr $\rightarrow$ cell
	upd	:	mem $\times$ addr $\times$ cell $\rightarrow$ mem
Signature:	Σ_{reach}		
Sorts:	mem, addr, $\mathsf{level_K}$, path		
Functions:	ϵ	:	path
	[_]	:	addr $\rightarrow$ path
Predicates:	*append*	:	path $\times$ path $\times$ path
	$\mathit{reach}_{\mathsf{K}}$	:	mem $\times$ addr $\times$ addr $\times$ $\mathsf{level_K} \times$ path
Signature:	Σ_{set}		
Sorts:	addr, set		
Functions:	$\emptyset$	:	set
	{_}	:	addr $\rightarrow$ set
	$\cup, \cap, \setminus$	:	set $\times$ set $\rightarrow$ set
Predicates:	$\in$	:	addr $\times$ set
	$\subseteq$	:	set $\times$ set

Figure 5.1: The signature of the $\mathsf{TSL_K}$ theory - Part I

Signature:	Σ_{settid}		
Sorts:	thid, settid		
Functions:	$\emptyset_{\mathrm{T}}$	:	settid
	$\{\_\}_{\mathrm{T}}$	:	thid $\rightarrow$ settid
	$\cup_{\mathrm{T}}, \cap_{\mathrm{T}}, \setminus_{\mathrm{T}}$	:	settid $\times$ settid $\rightarrow$ settid
Predicates:	$\in_{\mathrm{T}}$	:	thid $\times$ settid
	$\subseteq_{\mathrm{T}}$	:	settid $\times$ settid
Signature:	Σ_{mrgn}		
Sorts:	mrgn, addr, $\mathsf{level}_{\mathsf{K}}$		
Functions:	$\mathbf{emp}_{\mathrm{MR}}$	:	mrgn
	$\langle \_, \_ \rangle_{\mathrm{MR}}$	:	addr $\times$ $\mathsf{level}_{\mathsf{K}}$ $\rightarrow$ mrgn
	$\cup_{\mathrm{MR}}, \cap_{\mathrm{MR}}, -_{\mathrm{MR}}$	:	mrgn $\times$ mrgn $\rightarrow$ mrgn
Predicates:	$\in_{\mathrm{MR}}$	:	addr $\times$ $\mathsf{level}_{\mathsf{K}}$ $\times$ mrgn
	$\subseteq_{\mathrm{MR}}$	:	mrgn $\times$ mrgn
	$\#_{\mathrm{MR}}$	:	mrgn $\times$ mrgn
Signature:	Σ_{aarr}		
Sorts:	aarr, thid, addr		
Functions:	$\_[\_]_{\mathrm{A}}$	:	aarr $\times$ thid $\rightarrow$ addr
	$\_\{\_ \leftarrow \_\}_{\mathrm{A}}$	:	aarr $\times$ thid $\times$ addr $\rightarrow$ aarr
Signature:	Σ_{larr}		
Sorts:	larr, thid, $\mathsf{level}_{\mathsf{K}}$		
Functions:	$\_[\_]_{\mathrm{L}}$	:	larr $\times$ thid $\rightarrow$ $\mathsf{level}_{\mathsf{K}}$
	$\_\{\_ \leftarrow \_\}_{\mathrm{L}}$	:	larr $\times$ thid $\times$ $\mathsf{level}_{\mathsf{K}}$ $\rightarrow$ larr
Signature:	Σ_{bridge}		
Sorts:	mem, addr, $\mathsf{level}_{\mathsf{K}}$, set, path		
Functions:	*path2set*	:	path $\rightarrow$ set
	$addr2set_{\mathsf{K}}$	:	mem $\times$ addr $\times$ $\mathsf{level}_{\mathsf{K}}$ $\rightarrow$ set
	$getp_{\mathsf{K}}$	:	mem $\times$ addr $\times$ addr $\times$ $\mathsf{level}_{\mathsf{K}}$ $\rightarrow$ path
	$fstlock_{\mathsf{K}}$	:	mem $\times$ path $\times$ $\mathsf{level}_{\mathsf{K}}$ $\rightarrow$ addr
Predicates:	*ordList*	:	mem $\times$ path

Figure 5.2: The signature of the $\mathsf{TSL}_{\mathsf{K}}$ theory - Part II

signature: Σ_{cell}

interpretation:

- $mkcell^{\mathcal{A}}(e, k, \overrightarrow{a}, \overrightarrow{t}) = \langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle$
- $error^{\mathcal{A}}.next^{\mathcal{A}} = null^{\mathcal{A}}$
- $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.data^{\mathcal{A}} = e$
- $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.key^{\mathcal{A}} = k$
- $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.next^{\mathcal{A}}[j] = a_j$
- $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.lockid^{\mathcal{A}}[j] = t_j$
- $\langle e, k, \overrightarrow{a}, ...t_{j-1}, t_j, t_{j+1}... \rangle.lock^{\mathcal{A}}[j](t') = \langle e, k, \overrightarrow{a}, ...t_{j-1}, t', t_{j+1}... \rangle$
- $\langle e, k, \overrightarrow{a}, ...t_{j-1}, t_j, t_{j+1}... \rangle.unlock^{\mathcal{A}}[j] = \langle e, k, \overrightarrow{a}, ...t_{j-1}, \oslash, t_{j+1}... \rangle$

for each $e \in \mathcal{A}_{\mathsf{elem}}$, $k \in \mathcal{A}_{\mathsf{ord}}$, $t_0, \ldots, t_j, t_{j+1}, t_{j-1}, t' \in \mathcal{A}_{\mathsf{thid}}$, $\overrightarrow{a} \in \mathcal{A}^{\mathsf{K}}_{\mathsf{addr}}$, $\overrightarrow{t} \in \mathcal{A}^{\mathsf{K}}_{\mathsf{thid}}$ and $j \in \mathcal{A}_{\mathsf{level_K}}$

signature: Σ_{mem}

interpretation:

- $m[a]^{\mathcal{A}} = m(a)$
- $upd^{\mathcal{A}}(m, a, c) = m_{a \mapsto c}$
- $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$

for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$

signature: Σ_{reach}

interpretation:

- $\epsilon^{\mathcal{A}}$ is the empty sequence
- $[i]^{\mathcal{A}}$ is the sequence containing $i \in \mathcal{A}_{\mathsf{addr}}$ as the only element
- $([i_1 \,..\, i_n], [j_1 \,..\, j_m], [i_1 \,..\, i_n, j_1 \,..\, j_m]) \in append^{\mathcal{A}}$ iff $i_k \neq j_l$
- $(m, a_{init}, a_{end}, l, p) \in reach_{\mathsf{K}}{}^{\mathcal{A}}$ iff $a_{init} = a_{end}$ and $p = \epsilon$, or there exist addresses $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that:
  - (a) $p = [a_1 \,..\, a_n]$
  - (b) $a_1 = a_{init}$
  - (c) $m(a_r).next^{\mathcal{A}}[l] = a_{r+1}$, for $r < n$
  - (d) $m(a_n).next^{\mathcal{A}}[l] = a_{end}$

signature: Σ_{mrgn}

interpretation:

- $\mathbf{emp}^{\mathcal{A}}_{\mathrm{MR}} = \emptyset$
- $r \cup^{\mathcal{A}}_{\mathrm{MR}} s = r \cup s$
- $(a, j) \in^{\mathcal{A}}_{\mathrm{MR}} r \leftrightarrow (a, j) \in r$
- $\langle a, j \rangle^{\mathcal{A}}_{\mathrm{MR}} = \{(a, j)\}$
- $r \cap^{\mathcal{A}}_{\mathrm{MR}} s = r \cap s$
- $r \subseteq^{\mathcal{A}}_{\mathrm{MR}} s \leftrightarrow r \subseteq s$
- $r -^{\mathcal{A}}_{\mathrm{MR}} s = r \setminus s$
- $r \#^{\mathcal{A}}_{\mathrm{MR}} s \leftrightarrow r \cap^{\mathcal{A}}_{\mathrm{MR}} s = \mathbf{emp}^{\mathcal{A}}_{\mathrm{MR}}$

for each $a \in \mathcal{A}_{\mathsf{addr}}$, $j \in \mathcal{A}_{\mathsf{level_K}}$ and $r, s \in \mathcal{A}_{\mathsf{mrgn}}$

signature: Σ_{aarr}

interpretation:

- $x[t]^{\mathcal{A}}_{\mathrm{A}} = x(t)$
- $x\{t \leftarrow a\}^{\mathcal{A}}_{\mathrm{A}} = x_{new}$, such that $x_{new}(i) = \begin{cases} a & \text{if } i = t \\ x(i) & \text{otherwise} \end{cases}$

for each $x, x_{new} \in \mathcal{A}_{\mathsf{aarr}}$, $t \in \mathcal{A}_{\mathsf{thid}}$ and $a \in \mathcal{A}_{\mathsf{addr}}$

signature: Σ_{larr}

interpretation:

- $z[t]^{\mathcal{A}}_{\mathrm{L}} = x(t)$
- $z\{t \leftarrow l\}^{\mathcal{A}}_{\mathrm{L}} = z_{new}$, such that $z_{new}(i) = \begin{cases} l & \text{if } i = t \\ z(i) & \text{otherwise} \end{cases}$

for each $z, z_{new} \in \mathcal{A}_{\mathsf{larr}}$, $t \in \mathcal{A}_{\mathsf{thid}}$ and $l \in \mathcal{A}_{\mathsf{level_K}}$

Figure 5.3: Characterization of a TSL$_K$-interpretation $\mathcal{A}$

signature: Σ_{bridge}

interpretation:

- $path2set^{\mathcal{A}}(p) = \{a_1, \ldots, a_n\}$ for $p = [a_1, \ldots, a_n] \in \mathcal{A}_{\mathsf{path}}$
- $addr2set_{\mathsf{K}}{}^{\mathcal{A}}(m, a, l) = \left\{a' \in \mathcal{A}_{\mathsf{addr}} \mid \exists p \in \mathcal{A}_{\mathsf{path}} \,.\, (m, a, a', l, p) \in reach_{\mathsf{K}}\right\}$
- $getp_{\mathsf{K}}{}^{\mathcal{A}}(m, a_{init}, a_{end}, l) = \begin{cases} p & \text{if } (m, a_{init}, a_{end}, l, p) \in reach_{\mathsf{K}}{}^{\mathcal{A}} \\ \epsilon & \text{otherwise} \end{cases}$

  for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$, $l \in \mathcal{A}_{\mathsf{level_K}}$ and $a_{init}, a_{end} \in \mathcal{A}_{\mathsf{addr}}$
- $fstlock^{\mathcal{A}}(m, [a_1 \,..\, a_n], l) = \begin{cases} a_k & \text{if there is } k \leq n \text{ such that} \\ & \text{for all } j < k, m[a_j].lockid[l] = \oslash \\ & \text{and } m[a_k].lockid[l] \neq \oslash \\ null & \text{otherwise} \end{cases}$
- $ordList^{\mathcal{A}}(m, p)$ iff
  - $p = \epsilon$, or
  - $p = [a]$, or
  - $p = [a_1 \,..\, a_n]$ with $n \geq 2$ and $m(a_i).key^{\mathcal{A}} \preceq m(a_{i+1}).key^{\mathcal{A}}$ for all $1 \leq i < n$, for any $m \in \mathcal{A}_{\mathsf{mem}}$

Figure 5.4: Characterization of a $\mathsf{TSL_K}$-interpretation $\mathcal{A}$

interpretation over sets of addresses and thread identifiers respectively. Each sort σ in $\Sigma_{\mathsf{TSL_K}}$ is mapped to a non-empty set $\mathcal{A}_\sigma$ such that:

(a) $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{elem}}$ are discrete sets.

(b) $\mathcal{A}_{\mathsf{thid}}$ is a discrete set containing $\oslash$.

(c) $\mathcal{A}_{\mathsf{level_K}}$ is the finite collection $0, \ldots, \mathsf{K} - 1$.

(d) $\mathcal{A}_{\mathsf{ord}}$ is a total ordered set.

(e) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{ord}} \times \mathcal{A}_{\mathsf{addr}}^{\mathsf{K}} \times \mathcal{A}_{\mathsf{thid}}^{\mathsf{K}}$.

(f) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$.

(g) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$.

(h) $\mathcal{A}_{\mathsf{set}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$.

(i) $\mathcal{A}_{\mathsf{settid}}$ is the power-set of $\mathcal{A}_{\mathsf{thid}}$.

(j) $\mathcal{A}_{\mathsf{mrgn}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{level_K}}$.

(k) $\mathcal{A}_{\mathsf{aarr}} = \mathcal{A}_{\mathsf{addr}}^{\mathcal{A}_{\mathsf{thid}}}$.

(l) $\mathcal{A}_{\mathsf{larr}} = \mathcal{A}_{\mathsf{level_K}}^{\mathcal{A}_{\mathsf{thid}}}$.

Informally, sort addr represents addresses; elem the universe of elements that can be stored in the skiplist; ord the ordered keys used to preserve a strict order in the skiplist; thid thread identifiers; $\mathsf{level_K}$ the levels of a skiplist; cell models *cells* representing a node in a skiplist; mem models the heap, mapping addresses to cells; path describes finite sequences of non-repeating addresses to model non-cyclic list paths; set models sets of addresses – also known as regions –, while settid models sets of thread identifiers and mrgn masked regions; aarr models arrays of addresses indexed by thread identifiers and finally larr models arrays of skiplist levels indexed by thread identifiers.

$\Sigma_{\mathsf{level_K}}$ contains symbols for level identifiers $0, 1, \ldots, \mathsf{K} - 1$ and their conventional order. Σ_{ord} contains two special elements $-\infty$ and ∞ for the lowest and highest values in the order $\preceq$. Σ_{thid} only contains,

besides = and ≠ as for all the other theories, a special constant ⊘ to represent the absence of a thread identifier. Σ_{cell} contains the constructors and selectors for building and inspecting cells, including *error* for incorrect dereferences. Σ_{mem} is the signature for heaps, with the usual memory access and single memory mutation functions. Σ_{set} and Σ_{settid} are the signatures of theories of sets of addresses and thread ids resp. Σ_{mrgn} is the signature of the theory of masked regions while Σ_{aarr} and Σ_{larr} are the signatures for the theory of arrays over addresses and skiplist levels respectively, indexed in both cases by thread identifiers. The signature Σ_{reach} contains predicates to check reachability of address using paths at different levels, while Σ_{bridge} contains auxiliary functions and predicates to manipulate and inspect paths and locks.

Example 5.1
Consider the skiplist shown in Fig. 3.3 in Section 3.2. For that particular list and assuming that thread id variable j is assigned to value T_1, we can construct a model $\mathcal{A}$ such that for instance:

$$\begin{aligned}
\mathcal{A}_{\mathsf{addr}} &= \{0\text{x}00, 0\text{x}01, 0\text{x}02, 0\text{x}03, 0\text{x}04, 0\text{x}05, 0\text{x}06, 0\text{x}07, 0\text{x}08\} \\
\mathcal{A}_{\mathsf{ord}} = \mathcal{A}_{\mathsf{elem}} &= \{-\infty, 2, 8, 11, 14, 16, 20, +\infty\} \\
\mathcal{A}_{\mathsf{thid}} &= \{T_1, T_2, \oslash\} \\
\mathcal{A}_{\mathsf{level_K}} &= \{0, 1, 2, 3\} \\
\mathcal{A}_{\mathsf{cell}} &= \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{ord}} \times \mathcal{A}^4_{\mathsf{addr}} \times \mathcal{A}^4_{\mathsf{thid}} \\
\\
\mathcal{A}_{\mathsf{mem}} &= \{m : \mathcal{A}_{\mathsf{addr}} \rightarrow \mathcal{A}_{\mathsf{cell}}\}
\end{aligned}$$

where

$$\begin{aligned}
\mathit{null}^{\mathcal{A}} &= 0\text{x}00 \\
\mathit{error}^{\mathcal{A}} &= \langle\ -\infty\ ,\ -\infty\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ \rangle \\
\\
m(0\text{x}00) &= \langle\ -\infty\ ,\ -\infty\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ \rangle \\
m(0\text{x}01) &= \langle\ -\infty\ ,\ -\infty\ ,\ 0\text{x}02\ ,\ 0\text{x}02\ ,\ 0\text{x}03\ ,\ 0\text{x}08\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ \rangle \\
m(0\text{x}02) &= \langle\ 2\ ,\ 2\ ,\ 0\text{x}03\ ,\ 0\text{x}03\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ \rangle \\
m(0\text{x}03) &= \langle\ 8\ ,\ 8\ ,\ 0\text{x}04\ ,\ 0\text{x}05\ ,\ 0\text{x}06\ ,\ \mathit{null}\ ,\ \oslash\ ,\ T_1\ ,\ T_1\ ,\ \oslash\ \rangle \\
m(0\text{x}04) &= \langle\ 11\ ,\ 11\ ,\ 0\text{x}05\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ T_1\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ \rangle \\
m(0\text{x}05) &= \langle\ 14\ ,\ 14\ ,\ 0\text{x}06\ ,\ 0\text{x}06\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ T_1\ ,\ T_1\ ,\ \oslash\ ,\ \oslash\ \rangle \\
m(0\text{x}06) &= \langle\ 16\ ,\ 16\ ,\ 0\text{x}07\ ,\ 0\text{x}08\ ,\ 0\text{x}08\ ,\ \mathit{null}\ ,\ \oslash\ ,\ \oslash\ ,\ T_1\ ,\ \oslash\ \rangle \\
m(0\text{x}07) &= \langle\ 20\ ,\ 20\ ,\ 0\text{x}08\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ \rangle \\
m(0\text{x}08) &= \langle\ +\infty\ ,\ +\infty\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \mathit{null}\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ ,\ \oslash\ \rangle
\end{aligned}$$

and where for instance the masked region depicted in the figure is described by the set

$$\{(0\text{x}03, 1), (0\text{x}03, 2), (0\text{x}04, 0), (0\text{x}05, 0), (0\text{x}05, 1), (0\text{x}06, 2)\}$$ *

Once more, we are interested in analyzing the satisfiability of quantifier-free first order formulas. Hence, if φ is a formula, we first proceed to write it into its disjunctive normal form, let's say $\varphi_1 \vee \cdots \vee \varphi_n$. Then, we just verify the satisfiability of any of the φ_i, where each φ_i is a conjunction of $\mathsf{TSL_K}$ literals. We now classify the $\mathsf{TSL_K}$ literals into normalized and non-normalized ones. Non-normalized literals have the property that they can be written in terms of normalized ones. We now define the set of normalized $\mathsf{TSL_K}$-literals.

Definition 5.1 ($\mathsf{TSL_K}$-normalized literals).
A TSL_K-literal is normalized if it is a flat literal of the form:

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$a = null$	$c = error$	$c = rd(m, a)$
$k_1 \neq k_2$	$k_1 \preceq k_2$	$m_2 = upd(m_1, a, c)$
$c = mkcell(e, k, a_0, \ldots, a_{K-1}, t_0, \ldots, t_{K-1})$		$l_1 < l_2$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$g = \{t\}_{\mathrm{T}}$	$g_1 = g_2 \cup_{\mathrm{T}} g_3$	$g_1 = g_2 \setminus_{\mathrm{T}} g_3$
$r = \langle a, l \rangle_{\mathrm{MR}}$	$r_1 = r_2 \cup_{\mathrm{MR}} r_3$	$r_1 = r_2 -_{\mathrm{MR}} r_3$
$a = x[t]_{\mathrm{A}}$	$x_{new} = x\{t \leftarrow a\}_{\mathrm{A}}$	
$l = z[t]_{\mathrm{L}}$	$z_{new} = z\{t \leftarrow l\}_{\mathrm{L}}$	
$p_1 \neq p_2$	$p = [a]$	$p_1 = rev(p_2)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$
$s = addr2set_K(m, a, l)$	$p = getp_K(m, a_1, a_2, l)$	
$t_1 \neq t_2$	$a = fstlock\,(m, p, l)$	$ordList(m, p)$

where e, e_1 and e_2 are elem*-variables; a, a_0, a_1, a_2, $\ldots$, a_{K-1} are* addr*-variables; c is a* cell*-variable; m, m_1 and m_2 are* mem*-variables; p, p_1, p_2 and p_3 are* path*-variables; s, s_1, s_2 and s_3 are* set*-variables; g, g_1, g_2 and g_3 are* settid*-variables; r, r_1, r_2 and r_3 are* mrgn*-variables; x and x_{new} are* aarr*-variables; z and z_{new} are Σ_{larr}-variables; k, k_1 and k_2 are* ord*-variables; l, l_1 and l_2 are* $\mathsf{level_K}$*-variables and t, t_0, t_1, t_2, $\ldots$, t_{K-1} are* thid*-variables.* †

The remaining literals can be rewritten from the normalized ones using the following equivalences:

$$
\begin{array}{lcll}
e = c.data & \leftrightarrow & (\exists_{\mathsf{ord}} k\ \exists_{\mathsf{addr}} a_0, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{\mathsf{K}-1}) & \\
 & & \quad [c = mkcell\,(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})] & \\
k = c.key & \leftrightarrow & (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a_0, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{\mathsf{K}-1}) & \\
 & & \quad [c = mkcell\,(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})] & \\
a = c.next[l] & \leftrightarrow & (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{ord}} k\ \exists_{\mathsf{addr}} a_0, \ldots, a_{l-1}, a_{l+1}, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{\mathsf{K}-1}) & \\
 & & \quad [c = mkcell\,(e, k, a_0, \ldots, a_{l-1}, a, a_{l+1}, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})] & \\
t = c.lockid[l] & \leftrightarrow & (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{ord}} k\ \exists_{\mathsf{addr}} a_0, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{l-1}, t_{l+1}, \ldots, t_{\mathsf{K}-1}) & \\
 & & \quad [c = mkcell\,(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{l-1}, t, t_{l+1}, \ldots, t_{\mathsf{K}-1})] & \\
c_1 = c_2.lock[l](t) & \leftrightarrow & c_2.data = c_1.data \wedge c_2.key = c_1.key & \wedge \\
 & & c_2.next[0] = c_1.next[0]\ \wedge \cdots \wedge\ c_2.next[\mathsf{K}-1] = c_1.next[\mathsf{K}-1] & \wedge \\
 & & c_2.lockid[0] = c_1.lockid[0]\ \wedge \cdots \wedge\ c_2.lockid[l-1] = c_1.lockid[l-1] & \wedge \\
 & & c_2.lockid[l+1] = c_1.lockid[l+1]\ \wedge \cdots \wedge\ c_2.lockid[\mathsf{K}-1] = c_1.lockid[\mathsf{K}-1] & \wedge \\
 & & t = c_1.lockid[l] & \\
c_1 = c_2.unlock(l) & \leftrightarrow & c_2.data = c_1.data \wedge c_2.key = c_1.key & \wedge \\
 & & c_2.next[0] = c_1.next[0]\ \wedge \cdots \wedge\ c_2.next[\mathsf{K}-1] = c_1.next[\mathsf{K}-1] & \wedge \\
 & & c_2.lockid[0] = c_1.lockid[0]\ \wedge \cdots \wedge\ c_2.lockid[l-1] = c_1.lockid[l-1] & \wedge \\
 & & c_2.lockid[l+1] = c_1.lockid[l+1]\ \wedge \cdots \wedge\ c_2.lockid[\mathsf{K}-1] = c_1.lockid[\mathsf{K}-1] & \wedge \\
 & & \oslash = c_1.lockid[l] & \\
c_1 \neq_{\mathsf{cell}} c_2 & \leftrightarrow & c_1.data \neq c_2.data \vee c_1.key \neq c_2.key & \vee \\
 & & c_1.next[0] \neq c_2.next[0]\ \vee \ldots \vee\ c_1.next[\mathsf{K}-1] \neq c_2.next[\mathsf{K}-1] & \vee \\
 & & c_1.lockid[0] \neq c_2.lockid[0]\ \vee \cdots \vee\ c_1.lockid[\mathsf{K}-1] \neq c_2.lockid[\mathsf{K}-1] & \\
m_1 \neq_{\mathsf{mem}} m_2 & \leftrightarrow & (\exists_{\mathsf{addr}} a)\,[rd(m_1, a) \neq rd(m_2, a)] &
\end{array}
$$

$$
\begin{array}{lcl}
s_1 \neq s_2 & \leftrightarrow & (\exists_{\mathsf{addr}} a)\,[a \in (s_1 \setminus s_2) \cup (s_2 \setminus s_1)] \\
s = \emptyset & \leftrightarrow & s = s \setminus s \\
s_3 = s_1 \cap s_2 & \leftrightarrow & s_3 = (s_1 \cup s_2) \setminus ((s_1 \setminus s_2) \cup (s_2 \setminus s_1)) \\
a \in s & \leftrightarrow & \{a\} \subseteq s \\
s_1 \subseteq s_2 & \leftrightarrow & s_2 = s_1 \cup s_2 \\
g_1 \neq_{\mathsf{settid}} g_2 & \leftrightarrow & (\exists_{\mathsf{thid}} t)\,[t \in (g_1 \setminus_{\mathrm{T}} g_2) \cup_{\mathrm{T}} (g_2 \setminus_{\mathrm{T}} g_1)] \\
g = \emptyset_{\mathrm{T}} & \leftrightarrow & g = g \setminus_{\mathrm{T}} g \\
g_3 = g_1 \cap_{\mathrm{T}} g_2 & \leftrightarrow & g_3 = (g_1 \cup_{\mathrm{T}} g_2) \setminus_{\mathrm{T}} ((g_1 \setminus_{\mathrm{T}} g_2) \cup_{\mathrm{T}} (g_2 \setminus_{\mathrm{T}} g_1)) \\
t \in_{\mathrm{T}} g & \leftrightarrow & \{t\}_{\mathrm{T}} \subseteq_{\mathrm{T}} g \\
g_1 \subseteq_{\mathrm{T}} g_2 & \leftrightarrow & g_2 = g_1 \cup_{\mathrm{T}} g_2 \\
r_1 \neq_{\mathsf{settid}} r_2 & \leftrightarrow & (\exists_{\mathsf{addr}} a\ \exists_{\mathsf{level_K}} l)\,[(a,l) \in (r_1 -_{\mathrm{MR}} r_2) \cup_{\mathrm{MR}} (r_2 -_{\mathrm{MR}} r_1)] \\
r = \mathbf{emp}_{\mathrm{MR}} & \leftrightarrow & r = r -_{\mathrm{MR}} r \\
r_3 = r_1 \cap_{\mathrm{MR}} r_2 & \leftrightarrow & r_3 = (r_1 \cup_{\mathrm{MR}} r_2) -_{\mathrm{MR}} ((r_1 -_{\mathrm{MR}} r_2) \cup_{\mathrm{MR}} (r_2 -_{\mathrm{MR}} r_1)) \\
(a,l) \in_{\mathrm{MR}} r & \leftrightarrow & \langle a,l \rangle_{\mathrm{MR}} \subseteq_{\mathrm{MR}} r \\
r_1 \subseteq_{\mathrm{MR}} r_2 & \leftrightarrow & r_2 = r_1 \cup_{\mathrm{MR}} r_2 \\
r_1 \#_{\mathrm{MR}} r_2 & \leftrightarrow & \mathbf{emp}_{\mathrm{MR}} = (r_1 \cup_{\mathrm{MR}} r_2) -_{\mathrm{MR}} ((r_1 -_{\mathrm{MR}} r_2) \cup_{\mathrm{MR}} (r_2 -_{\mathrm{MR}} r_1)) \\
p = \epsilon & \leftrightarrow & \mathit{append}(p,p,p) \\
\mathit{reach}_{\mathsf{K}}(m,a_1,a_2,l,p) & \leftrightarrow & a_2 \in \mathit{addr2set}_{\mathsf{K}}(m,a_1,l) \wedge p = \mathit{getp}_{\mathsf{K}}(m,a_1,a_2,l)
\end{array}
$$

This means that we can rewrite such literals using:

Flat:	$e = c.\mathit{data}$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{\mathsf{K}-1})$
Proviso:	$k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$k = c.\mathit{key}$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{\mathsf{K}-1})$
Proviso:	$e,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$a = c.\mathit{next}[l]$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\ldots,a_{l-1},a,a_{l+1},\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\ldots,a_{l-1},a_{l+1},a_{\mathsf{K}-1},t_0,\ldots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$t = c.\mathit{lockid}[l]$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},t,t_{l+1},\ldots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},t_{l+1},\ldots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$c_1 = c_2.\mathit{lock}[l](t)$
Normalized:	$c_1 = \mathit{mkcell}(e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},t,t_{l+1},\ldots,t_{\mathsf{K}-1}) \wedge$
	$c_2 = \mathit{mkcell}(e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},\tilde{t},t_{l+1},\ldots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},\tilde{t},t_{l+1},\ldots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$c_1 = c_2.\mathit{unlock}[l]$
Normalized:	$c_1 = \mathit{mkcell}(e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},\oslash,t_{l+1},\ldots,t_{\mathsf{K}-1}) \wedge$
	$c_2 = \mathit{mkcell}(e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},\tilde{t},t_{l+1},\ldots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\ldots,a_{\mathsf{K}-1},t_0,\ldots,t_{l-1},\tilde{t},t_{l+1},\ldots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$c_1 \neq c_2$
Normalized:	$c_1.\mathit{data} \neq c_2.\mathit{data} \vee c_1.\mathit{key} \neq c_2.\mathit{key} \vee$
	$c_1.\mathit{next}[0] \neq c_2.\mathit{next}[0] \vee \cdots \vee c_1.\mathit{next}[\mathsf{K}-1] \neq c_2.\mathit{next}[\mathsf{K}-1] \vee$
	$c_1.\mathit{lockid}[0] \neq c_2.\mathit{lockid}[0] \vee \cdots \vee c_1.\mathit{lockid}[\mathsf{K}-1] \neq c_2.\mathit{lockid}[\mathsf{K}-1]$
Proviso:	-

Flat:	$m_1 \neq m_2$
Normalized:	$m[a] \neq m[a]$
Proviso:	a is fresh.
Flat:	$s_1 \neq s_2$
Normalized:	$s_{12} = s_1 \setminus s_2 \wedge s_{21} = s_2 \setminus s_1 \wedge s_3 = s_{12} \cup s_{21} \wedge s = s_3 \cup \{a\} \wedge \{a\} \subseteq s$
Proviso:	s_{12}, s_{21}, s_3, s and a are fresh.
Flat:	$s = \emptyset$
Normalized:	$s = s \setminus s$
Proviso:	-
Flat:	$s_3 = s_1 \cap s_2$
Normalized:	$s_{12} = s_1 \setminus s_2 \wedge s_{21} = s_2 \setminus s_1 \wedge s_{u_1} = s_1 \cup s_2 \wedge s_{u_2} = s_{12} \cup s_{21} \wedge$ $s_3 = s_{u_1} \setminus s_{u_2}$
Proviso:	s_{12}, s_{21}, s_{u_1} and s_{u_2} are fresh.
Flat:	$a \in s$
Normalized:	$s = \{a\} \cup s$
Proviso:	-
Flat:	$s_1 \subseteq s_2$
Normalized:	$s_2 = s_1 \cup s_2$
Proviso:	-
Flat:	$g_1 \neq g_2$
Normalized:	$g_{12} = g_1 \setminus_{\mathrm{T}} g_2 \wedge g_{21} = g_2 \setminus_{\mathrm{T}} g_1 \wedge g_3 = g_{12} \cup_{\mathrm{T}} g_{21} \wedge g = g_3 \cup_{\mathrm{T}} \{t\} \wedge \{t\} \subseteq_{\mathrm{T}} g$
Proviso:	g_{12}, g_{21}, g_3, g and t are fresh.
Flat:	$g = \emptyset_{\mathrm{T}}$
Normalized:	$g = g \setminus_{\mathrm{T}} g$
Proviso:	-
Flat:	$g_3 = g_1 \cap_{\mathrm{T}} g_2$
Normalized:	$g_{12} = g_1 \setminus_{\mathrm{T}} g_2 \wedge g_{21} = g_2 \setminus_{\mathrm{T}} g_1 \wedge g_{u_1} = g_1 \cup_{\mathrm{T}} g_2 \wedge g_{u_2} = g_{12} \cup_{\mathrm{T}} g_{21} \wedge$ $g_3 = g_{u_1} \setminus_{\mathrm{T}} g_{u_2}$
Proviso:	g_{12}, g_{21}, g_{u_1} and g_{u_2} are fresh.
Flat:	$t \in_{\mathrm{T}} g$
Normalized:	$g = \{t\} \cup_{\mathrm{T}} g$
Proviso:	-
Flat:	$g_1 \subseteq_{\mathrm{T}} g_2$
Normalized:	$g_2 = g_1 \cup_{\mathrm{T}} g_2$
Proviso:	-
Flat:	$r_1 \neq r_2$
Normalized:	$r_{12} = r_1 -_{\mathrm{MR}} r_2 \wedge r_{21} = r_2 -_{\mathrm{MR}} r_1 \wedge r_3 = r_{12} \cup_{\mathrm{MR}} r_{21} \wedge$ $r = r_3 \cup_{\mathrm{MR}} \{(a, l)\} \wedge \{(a, l)\} \subseteq_{\mathrm{MR}} r$
Proviso:	$r_{12}, r_{21}, r_3, r, a$ and l are fresh.
Flat:	$r = \mathbf{emp}_{\mathrm{MR}}$
Normalized:	$r = r -_{\mathrm{MR}} r$
Proviso:	-
Flat:	$r_3 = r_1 \cap_{\mathrm{MR}} r_2$
Normalized:	$r_{12} = r_1 -_{\mathrm{MR}} r_2 \wedge r_{21} = r_2 -_{\mathrm{MR}} r_1 \wedge r_{u_1} = r_1 \cup_{\mathrm{MR}} r_2 \wedge$ $r_{u_2} = r_{12} \cup_{\mathrm{MR}} r_{21} \wedge r_3 = r_{u_1} -_{\mathrm{MR}} r_{u_2}$
Proviso:	r_{12}, r_{21}, r_{u_1} and r_{u_2} are fresh.
Flat:	$(a, l) \in_{\mathrm{MR}} r$
Normalized:	$r = \{(a, l)\} \cup_{\mathrm{MR}} r$
Proviso:	-
Flat:	$r_1 \subseteq_{\mathrm{MR}} r_2$
Normalized:	$r_2 = r_1 \cup_{\mathrm{MR}} r_2$
Proviso:	-

Flat:	$r_1 \#_{\text{MR}} r_2$
Normalized:	$r_{12} = r_1 -_{\text{MR}} r_2 \wedge r_{21} = r_2 -_{\text{MR}} r_1 \wedge r_{u_1} = r_1 \cup_{\text{MR}} r_2 \wedge$ $r_{u_2} = r_{12} \cup_{\text{MR}} r_{21} \wedge r_3 = r_{u_1} -_{\text{MR}} r_{u_2} \wedge r_3 = r_3 -_{\text{MR}} r_3$
Proviso:	$r_{12}, r_{21}, r_{u_1}, r_{u_2}$ and r_3 are fresh.
Flat:	$p = \epsilon$
Normalized:	$append(p, p, p)$
Proviso:	-
Flat:	$reach_{\mathsf{K}}(m, a_1, a_2, l, p)$
Normalized:	$a_2 \in addr2set_{\mathsf{K}}(m, a_1, l) \wedge p = getp_{\mathsf{K}}(m, a_1, a_2, l)$
Proviso:	-

5.1 Decidability of TSL$_{\mathsf{K}}$

We show that TSL$_{\mathsf{K}}$ is decidable by proving that it enjoys the finite model property with respect to its sorts, and exhibiting upper bounds for the sizes of the domains of a small interpretation of a satisfiable formula.

Definition 5.2 (Finite Model Property).
Let Σ be a signature, $S_0 \subseteq S$ be a set of sorts, and T be a Σ-theory. T has the finite model property with respect to S_0 if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ such that for each sort $\sigma \in S_0$, $\mathcal{A}_\sigma$ is finite. †

The fact that TSL$_{\mathsf{K}}$ has the finite model property with respect to domains elem, addr, ord, level$_{\mathsf{K}}$ and thid, implies that TSL$_{\mathsf{K}}$ is decidable by enumerating all possible $\Sigma_{\text{TSL}_{\mathsf{K}}}$-structures up to a certain cardinality. Notice that a bound on the domain of these sorts it is enough to get finite interpretations for the remaining sorts (cell, mem, path, set, settid, mrgn, aarr and larr) as the elements in the domains of these latter sorts are constructed using the elements in the domains of elem, addr, ord, level$_{\mathsf{K}}$ and thid.

Lemma 5.1:
Deciding the TSL$_K$-satisfiability of a quantifier-free TSL$_K$-formula is equivalent to verifying the TSL$_K$-satisfiability of the normalized TSL$_K$-literals. ♠

Proof First, transform a formula in disjunctive normal form. Then each conjunct can be normalized introducing auxiliary fresh variables when necessary. For instance, consider the non-normalized literal $c_1 = c_2.lock[l](t)$. According to the table we presented before, this literal can be written as a conjunction of normalized literals of the form

$$c_1 = mkcell(e, k, \overrightarrow{a}, t_0, \ldots, t_{l-1}, t, t_{l+1}, \ldots, t_{\mathsf{K}-1}) \wedge c_2 = mkcell(e, k, \overrightarrow{a}, t_0, \ldots, t_{\mathsf{K}-1})$$

where $e, k, \overrightarrow{a} = a_0 \ldots a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1}$ and t are fresh variables. Let $\mathcal{A}$ be a model satisfying the literal $c_1 = c_2.lock[l](t)$. Then, we show that

$$\mathcal{A} \vDash c_1 = c_2.lock[l](t)$$

iff

$$\mathcal{A} \vDash c_1 = mkcell(e, k, \overrightarrow{a}, t_0, \ldots, t_{l-1}, t, t_{l+1}, \ldots, t_{\mathsf{K}-1}) \wedge c_2 = mkcell(e, k, \overrightarrow{a}, t_0, \ldots, t_{\mathsf{K}-1})$$

The proof is as follows:

$$
\begin{array}{lll}
\mathcal{A} \models & c_1 = c_2.lock[l](t) & \leftrightarrow \\
& c_1^{\mathcal{A}} = c_2^{\mathcal{A}}.lock^{\mathcal{A}}[l^{\mathcal{A}}](t^{\mathcal{A}}) & \leftrightarrow \\
& c_1^{\mathcal{A}} = c_2^{\mathcal{A}}.lock^{\mathcal{A}}[l^{\mathcal{A}}](t^{\mathcal{A}}) \wedge c_2{}^{\mathcal{A}} = \langle e^{\mathcal{A}}, k^{\mathcal{A}}, \overrightarrow{a}^{\mathcal{A}}, t_0^{\mathcal{A}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}} \rangle & \leftrightarrow \\
& c_1^{\mathcal{A}} = \langle e^{\mathcal{A}}, k^{\mathcal{A}}, \overrightarrow{a}^{\mathcal{A}}, t_0^{\mathcal{A}}, \ldots, t_{l-1}^{\mathcal{A}}, t^{\mathcal{A}}, t_{l+1}^{\mathcal{A}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}} \rangle & \wedge \\
& c_2{}^{\mathcal{A}} = \langle e^{\mathcal{A}}, k^{\mathcal{A}}, \overrightarrow{a}^{\mathcal{A}}, t_0^{\mathcal{A}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}} \rangle & \leftrightarrow \\
& c_1^{\mathcal{A}} = mkcell^{\mathcal{A}}(e^{\mathcal{A}}, k^{\mathcal{A}}, \overrightarrow{a}^{\mathcal{A}}, t_0^{\mathcal{A}}, \ldots, t_{l-1}^{\mathcal{A}}, t^{\mathcal{A}}, t_{l+1}^{\mathcal{A}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}}) & \wedge \\
& c_2^{\mathcal{A}} = mkcell^{\mathcal{A}}(e^{\mathcal{A}}, k^{\mathcal{A}}, \overrightarrow{a}^{\mathcal{A}}, t_0^{\mathcal{A}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}}) & \leftrightarrow \\
\mathcal{A} \models & c_1 = mkcell(e, k, \overrightarrow{a}, t_0, \ldots, t_{l-1}, t, t_{l+1}, \ldots, t_{\mathsf{K}-1}) & \wedge \\
& c_2 = mkcell(e, k, \overrightarrow{a}, t_0, \ldots, t_{\mathsf{K}-1}) &
\end{array}
$$

The remaining cases can be proved in a similar way. □

The phase of normalizing a formula is commonly known [RRZ05] as the "variable abstraction phase". Note that each normalized literal belongs to just one of the theories that are part of TSL_K.

Consider an arbitrary TSL_K-interpretation $\mathcal{A}$ satisfying a conjunction of normalized TSL_K-literals Γ. We show that if $\mathcal{A}$ consists of domains $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$, $\mathcal{A}_{\mathsf{thid}}$, $\mathcal{A}_{\mathsf{level_K}}$ and $\mathcal{A}_{\mathsf{ord}}$ then there are finite sets $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{thid}}$, $\mathcal{B}_{\mathsf{level_K}}$ and $\mathcal{B}_{\mathsf{ord}}$ with bounded cardinalities, where the finite bound on the sizes can be computed from Γ. Such sets can in turn be used to obtain a finite interpretation $\mathcal{B}$ satisfying Γ, since all the other sorts are bounded by the sizes of these sets.

Before proving that TSL_K enjoys the finite model property, we define some auxiliary functions. We start by defining the function

$$first_\mathsf{K} \ : \ \mathsf{mem} \times \mathsf{addr} \times \mathsf{level_K} \times \mathsf{set} \rightarrow \mathsf{addr}$$

Let $\mathcal{B}_{\mathsf{addr}} \subseteq \mathcal{A}_{\mathsf{addr}}$, $m : \mathcal{A}_{\mathsf{addr}} \rightarrow \mathcal{B}_{\mathsf{elem}} \times \mathcal{B}_{\mathsf{ord}} \times \mathcal{A}_{\mathsf{addr}}^{\mathsf{K}} \times \mathcal{B}_{\mathsf{thid}}^{\mathsf{K}}$, $a \in \mathcal{B}_{\mathsf{addr}}$ and $l \in \mathcal{B}_{\mathsf{level_K}}$. The function $first_\mathsf{K}(m, a, l, \mathcal{B}_{\mathsf{addr}})$ is defined by

$$
first_\mathsf{K}(m, a, l, \mathcal{B}_{\mathsf{addr}}) \quad = \quad \begin{cases} null & \text{if for all } r \geq 1 \;\; m^r(a).next(l) \notin \mathcal{B}_{\mathsf{addr}} \\ m^s(a).next(l) & \text{if for some } s \geq 1 \;\; m^s(a).next(l) \in \mathcal{B}_{\mathsf{addr}}, \\ & \quad \text{and for all } r < s \;\; m^r(a).next(l) \notin \mathcal{B}_{\mathsf{addr}} \end{cases}
$$

where

- $m^1(a).next(l)$ stands for $m(a).next(l)$, and
- $m^{n+1}(a).next(l)$ stands for $m(m^n(a).next(l)).next(l)$ when $n > 0$.

Basically, given the original model $\mathcal{A}$ and a subset of addresses $\mathcal{B}_{\mathsf{addr}} \subseteq \mathcal{A}_{\mathsf{addr}}$, function $first_\mathsf{K}$ chooses the next address in $\mathcal{B}_{\mathsf{addr}}$ that can be reached from a given address following repeatedly the $next(l)$ pointer. It is easy to see, for example, that if $m(a).next(l) \in \mathcal{B}_{\mathsf{addr}}$ then $first_\mathsf{K}(m, a, l, \mathcal{B}_{\mathsf{addr}}) = m(a).next(l)$. We will later filter out unnecessary intermediate nodes and use $first_\mathsf{K}$ to bypass properly the removed nodes, preserving the important connectivity properties.

Lemma 5.2:
Let $\mathcal{B}_{\mathsf{addr}} \subseteq \mathcal{A}_{\mathsf{addr}}$, $m : \mathcal{A}_{\mathsf{addr}} \rightarrow \mathcal{B}_{\mathsf{elem}} \times \mathcal{B}_{\mathsf{ord}} \times \mathcal{A}_{\mathsf{addr}}^{K} \times \mathcal{B}_{\mathsf{thid}}^{K}$, $a \in \mathcal{B}_{\mathsf{addr}}$ and $l \in \mathcal{B}_{\mathsf{level_K}}$. Then, if $m(a).next(l) \in \mathcal{B}_{\mathsf{addr}}$, we have that $first_K(m, a, l, \mathcal{B}_{\mathsf{addr}}) = m(a).next(l)$. ♠

Proof Immediate from the definition of $first_{\mathsf{K}}$. □

Secondly, the function *unordered* that given a memory m and a path p, returns a set containing two addresses that witness the failure to preserve the *key* order of elements in p:

$$unordered(m, [i_1, \ldots, i_n]) = \begin{cases} \emptyset & \text{if } n = 0 \text{ or } n = 1 \\ \{i_1, i_2\} & \text{if } m(i_2).key \preceq m(i_1).key \text{ and } m(i_2).key \neq m(i_1).key \\ unordered(m, [i_2, \ldots, i_n]) & \text{otherwise} \end{cases}$$

If two such addresses exist, *unordered* returns the first two consecutive addresses whose keys violate the order.

Lemma 5.3:
Let p be a path such that $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and let m be a memory. If it exists a_i, with $1 \leq i < n$, such that $m(a_{i+1}).key \preceq m(a_i).key$ and $m(a_{i+1}).key \neq m(a_i).key$, then $unordered(m, p) \neq \emptyset$ ♠

Proof By induction. Let's consider $n = 2$ and let $p = [a_1, a_2]$ such that $m(a_2).key \preceq m(a_1).key$ and $m(a_2).key \neq m(a_1).key$. Then, by definition of *unordered*, we have that $unordered(m, p) = \{a_1, a_2\} \neq \emptyset$.

Now assume $n > 2$ and let $p = [a_1, \ldots, a_n]$. If $m(a_2).key \preceq m(a_1).key$ and $m(a_2).key \neq m(a_1).key$, then we have that $unordered(m, p) = \{a_1, a_2\} \neq \emptyset$. On the other hand, if $m(a_1).key \preceq m(a_2).key$, we still know that there is a_i, with $2 \leq i < n$, s.t., $m(a_{i+1}).key \preceq m(a_i).key$ and $m(a_{i+1}).key \neq m(a_i).key$. Therefore, by induction we have that $unordered(m, [a_2, \ldots, a_n]) \neq \emptyset$ and by definition of *unordered*, $unordered(m, p) = unordered(m, [a_2, \ldots, a_n]) \neq \emptyset$. □

Finally, we define the function *knownTID* that given a thread identifier and a set of threads, returns the same thread id if it belongs to the given set. Otherwise, it returns $\oslash$:

$$knownTID(t, T) = \begin{cases} t & \text{if } t \in T \\ \oslash & \text{otherwise} \end{cases}$$

In this section, we also make use of the functions *compress*, *diseq* and *common* defined in Section 4.1. We now show that given a conjunction of $\mathsf{TSL}_{\mathsf{K}}$ literals, if there exists a model satisfying such formula, then it is possible to construct a new model with a finite number of elements satisfying the same formula. This property is described in the following Lemma.

Lemma 5.4 (Finite Model Property):
Let Γ be a conjunction of normalized TSL_K-literals. Let $\overline{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\overline{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\overline{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\overline{p} = |V_{\mathsf{path}}(\Gamma)|$, $\overline{t} = |V_{\mathsf{thid}}(\Gamma)|$, $\overline{o} = |V_{\mathsf{ord}}(\Gamma)|$ and $\overline{x} = |V_{\mathsf{aarr}}(\Gamma)|$. Then the following are equivalent:

1. *Γ is TSL_K-satisfiable;*
2. *Γ is true in a TSL_K interpretation $\mathcal{B}$ such that*

$$\begin{aligned} |\mathcal{B}_{\mathsf{level_K}}| &\leq K \\ |\mathcal{B}_{\mathsf{thid}}| &\leq \overline{t} + 1 + K\,\overline{m}\,\overline{a} \\ |\mathcal{B}_{\mathsf{addr}}| &\leq \overline{a} + 1 + K\,\overline{m}\,\overline{a} + \overline{p}^2 + \overline{p}^3 + 2\,\overline{m}\,\overline{p} + \overline{x}\,|\mathcal{B}_{\mathsf{thid}}| \\ |\mathcal{B}_{\mathsf{elem}}| &\leq \overline{e} + \overline{m}\;|\mathcal{B}_{\mathsf{addr}}| \\ |\mathcal{B}_{\mathsf{ord}}| &\leq \overline{o} + \overline{m}\;|\mathcal{B}_{\mathsf{addr}}| \end{aligned}$$

♠

Proof $(2 \to 1)$ is immediate.

$(1 \to 2)$. We prove this implication only for the new TSL_K-literals since the ones shared with theory TLL3 have already been proved in Lemma 4.3.

Bearing in mind the auxiliary functions we have defined, let $\mathcal{A}$ be a TSL_K-interpretation satisfying a set of normalized TSL_K-literals Γ. We use $\mathcal{A}$ to construct a finite TSL_K-interpretation $\mathcal{B}$ that satisfies Γ.

$$\mathcal{B}_{\mathsf{level_K}} = \mathcal{A}_{\mathsf{level_K}} = [0 \ldots K-1]$$

$$\mathcal{B}_{\mathsf{thid}} = V_{\mathsf{thid}}^{\mathcal{A}} \cup \{\oslash\} \cup \left\{m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}(l) \mid m \in V_{\mathsf{mem}}, a \in V_{\mathsf{addr}} \text{ and } l \in \mathcal{B}_{\mathsf{level_K}}\right\}$$

$$\begin{aligned}
\mathcal{B}_{\mathsf{addr}} = {} & V_{\mathsf{addr}}^{\mathcal{A}} \cup \left\{null^{\mathcal{A}}\right\} \cup \\
& \left\{m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(l) \mid m \in V_{\mathsf{mem}}, a \in V_{\mathsf{addr}} \text{ and } l \in \mathcal{B}_{\mathsf{level_K}}\right\} \cup \\
& \left\{v \in diseq(p^{\mathcal{A}}, q^{\mathcal{A}}) \mid \text{ the literal } p \neq q \text{ is in } \Gamma\right\} \cup \\
& \left\{v \in common(p_1{}^{\mathcal{A}}, p_2{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and } path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) \neq \emptyset\right\} \cup \\
& \left\{v \in common(p_1{}^{\mathcal{A}} \circ p_2{}^{\mathcal{A}}, p_3{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and } path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) = \emptyset\right\} \cup \\
& \left\{v \in unordered(m^{\mathcal{A}}, p^{\mathcal{A}}) \mid \neg ordList(m, p) \text{ is in } \Gamma\right\} \cup \\
& \left\{x^{\mathcal{A}}(i) \mid x \in V_{\mathsf{aarr}} \text{ and } i \in \mathcal{B}_{\mathsf{thid}}\right\}
\end{aligned}$$

$$\mathcal{B}_{\mathsf{elem}} = V_{\mathsf{elem}}^{\mathcal{A}} \cup \left\{m^{\mathcal{A}}(v).data^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in \mathcal{B}_{\mathsf{addr}}\right\}$$

$$\mathcal{B}_{\mathsf{ord}} = V_{\mathsf{ord}}^{\mathcal{A}} \cup \left\{m^{\mathcal{A}}(v).key^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in \mathcal{B}_{\mathsf{addr}}\right\}$$

For each sort σ, $V_\sigma^{\mathcal{A}}$ denotes the set of values obtained by the interpretation of all variables of type σ appearing in Γ. Notice that as a finite number of variables can appear in Γ, $V_\sigma^{\mathcal{A}}$ is also a finite set.

The domains described above satisfy the cardinality constrains expressed in the statement of the theorem. The interpretations of the symbols are:

$$\begin{aligned}
error^{\mathcal{B}} &= error^{\mathcal{A}} \\
null^{\mathcal{B}} &= null^{\mathcal{A}} \\
e^{\mathcal{B}} &= e^{\mathcal{A}} && \text{for each } e \in V_{\mathsf{elem}} \\
a^{\mathcal{B}} &= a^{\mathcal{A}} && \text{for each } a \in V_{\mathsf{addr}} \\
c^{\mathcal{B}} &= c^{\mathcal{A}} && \text{for each } c \in V_{\mathsf{cell}} \\
t^{\mathcal{B}} &= t^{\mathcal{A}} && \text{for each } t \in V_{\mathsf{thid}} \\
k^{\mathcal{B}} &= k^{\mathcal{A}} && \text{for each } k \in V_{\mathsf{ord}} \\
l^{\mathcal{B}} &= l^{\mathcal{A}} && \text{for each } l \in V_{\mathsf{level_K}} \\
m^{\mathcal{B}}(v) &= \big(m^{\mathcal{A}}(v).data^{\mathcal{A}}, m^{\mathcal{A}}(v).key^{\mathcal{A}}, && \text{for each } m \in V_{\mathsf{mem}} \\
&\quad first_{\mathsf{K}}(m^{\mathcal{A}}, v, 0, \mathcal{B}_{\mathsf{addr}}), \ldots, first_{\mathsf{K}}(m^{\mathcal{A}}, v, \mathsf{K}-1, \mathcal{B}_{\mathsf{addr}}), && \text{and } v \in \mathcal{B}_{\mathsf{addr}} \\
&\quad knownTID(m^{\mathcal{A}}(v).lockid^{\mathcal{A}}(0), \mathcal{B}_{\mathsf{thid}}), \\
&\qquad \ldots, \\
&\quad knownTID(m^{\mathcal{A}}(v).lockid^{\mathcal{A}}(\mathsf{K}-1), \mathcal{B}_{\mathsf{thid}})\big) \\
s^{\mathcal{B}} &= s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} && \text{for each } s \in V_{\mathsf{set}} \\
g^{\mathcal{B}} &= g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} && \text{for each } g \in V_{\mathsf{settid}} \\
r^{\mathcal{B}} &= r^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}}) && \text{for each } r \in V_{\mathsf{mrgn}} \\
x^{\mathcal{B}}(i) &= x^{\mathcal{A}}(i) && \text{for each } x \in V_{\mathsf{aarr}}, i \in \mathcal{B}_{\mathsf{thid}} \\
z^{\mathcal{B}}(i) &= z^{\mathcal{A}}(i) && \text{for each } z \in V_{\mathsf{larr}}, i \in \mathcal{B}_{\mathsf{thid}} \\
p^{\mathcal{B}} &= compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) && \text{for each } p \in V_{\mathsf{path}}
\end{aligned}$$

Essentially, all variables and constants in $\mathcal{B}$ are interpreted as in $\mathcal{A}$ except that *next* pointers use $first_{\mathsf{K}}$ to point to the next reachable element that has been preserved in $\mathcal{B}_{\mathsf{addr}}$, and paths filter out all elements except those in $\mathcal{B}_{\mathsf{addr}}$. It can be routinely checked that $\mathcal{B}$ is an interpretation of Γ. So it remains to be seen that $\mathcal{B}$ satisfies all literals in Γ assuming that $\mathcal{A}$ does, concluding that $\mathcal{B}$ is indeed a model of Γ. This check is performed by cases. The proof that $\mathcal{B}$ satisfies all $\mathsf{TSL}_{\mathsf{K}}$-literals in Γ is not shown here. We just focus on the new functions and predicates that are not part of TLL. The proof for literals of TLL can be found in [RZ06a] and for TLL3 in Chapter 4. For $\mathsf{TSL}_{\mathsf{K}}$-literals we must consider the following cases:

Literals of the form $l_1 \neq l_2$, $k_1 \neq k_2$, $l_1 < l_2$ and $k_1 \preceq k_2$: Immediate.

Literals of the form $c = mkcell(e, k, a_0, \ldots, a_{\mathbf{K}-1}, t_0, \ldots, t_{\mathbf{K}-1})$:

$$\begin{aligned}
c^{\mathcal{B}} &= c^{\mathcal{A}} \\
&= mkcell^{\mathcal{A}}(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1}) \\
&= \langle e^{\mathcal{A}}, k^{\mathcal{A}}, a_0^{\mathcal{A}}, \ldots, a_{\mathsf{K}-1}^{\mathcal{A}}, t_0^{\mathcal{A}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}} \rangle \\
&= \langle e^{\mathcal{B}}, k^{\mathcal{B}}, a_0^{\mathcal{B}}, \ldots, a_{\mathsf{K}-1}^{\mathcal{B}}, t_0^{\mathcal{B}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{B}} \rangle \\
&= mkcell^{\mathcal{B}}(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})
\end{aligned}$$

Literals of the form $c = rd(m, a)$:

$$\begin{aligned}
\left[rd(m,a)\right]^{\mathcal{B}} &= m^{\mathcal{B}}\,(a^{\mathcal{B}}) \\
&= m^{\mathcal{B}}\,(a^{\mathcal{A}}) \\
&= \Big(m^{\mathcal{A}}\,(a^{\mathcal{A}}).data^{\mathcal{A}},\, m^{\mathcal{A}}\,(a^{\mathcal{A}}).key^{\mathcal{A}}, \\
&\qquad first_{\mathsf{K}}\,(m^{\mathcal{A}}, a^{\mathcal{A}}, 0, \mathcal{B}_{\mathsf{addr}}), \ldots, first_{\mathsf{K}}\,(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathsf{K}\text{-}1, \mathcal{B}_{\mathsf{addr}}), \\
&\qquad knownTID\,(m^{\mathcal{A}}\,(\mathrm{v}).lockid^{\mathcal{A}}\,(0), \mathcal{B}_{\mathsf{thid}}), \ldots, knownTID\,(m^{\mathcal{A}}\,(\mathrm{v}).lockid^{\mathcal{A}}\,(\mathsf{K}\text{-}1), \mathcal{B}_{\mathsf{thid}})\Big) \\
&= \Big(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, m^{\mathcal{A}}(a^{\mathcal{A}}).key^{\mathcal{A}}, \\
&\qquad m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(0), \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(\mathsf{K}-1), \\
&\qquad m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}[0], \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}[\mathsf{K}-1]\Big) \qquad \text{(Lemma 5.2)} \\
&= m^{\mathcal{A}}(a^{\mathcal{A}}) \\
&= c^{\mathcal{A}} \\
&= c^{\mathcal{B}}
\end{aligned}$$

Literals of the form $g = \{t\}_{\mathrm{T}}$:

$$g^{\mathcal{B}} = g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} = \{t^{\mathcal{A}}\}_{\mathrm{T}} \cap \mathcal{B}_{\mathsf{thid}} = \{t^{\mathcal{B}}\}_{\mathrm{T}} \cap \mathcal{B}_{\mathsf{thid}} = \{t^{\mathcal{B}}\}_{\mathrm{T}}$$

Literals of the form $g_1 = g_2 \cup_{\mathrm{T}} g_3$:

$$\begin{aligned}
g_1^{\mathcal{B}} &= g_1^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} \\
&= \left(g_2^{\mathcal{A}} \cup_{\mathrm{T}} g_3^{\mathcal{A}}\right) \cap \mathcal{B}_{\mathsf{thid}} \\
&= \left(g_2^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}}\right) \cup_{\mathrm{T}} \left(g_3^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}}\right) \\
&= g_2^{\mathcal{B}} \cup_{\mathrm{T}} g_3^{\mathcal{B}}
\end{aligned}$$

Literals of the form $g_1 = g_2 \setminus_{\mathbf{T}} g_3$:

$$\begin{aligned} g_1^{\mathcal{B}} &= g_1^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} \\ &= \left(g_2^{\mathcal{A}} \setminus_{\mathbf{T}} g_3^{\mathcal{A}}\right) \cap \mathcal{B}_{\mathsf{thid}} \\ &= \left(g_2^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}}\right) \setminus_{\mathbf{T}} \left(g_3^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}}\right) \\ &= g_2^{\mathcal{B}} \setminus_{\mathbf{T}} g_3^{\mathcal{B}} \end{aligned}$$

Literals of the form $r = \langle a, l \rangle_{\mathbf{MR}}$:

$$\begin{aligned} r^{\mathcal{B}} &= r^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}}) \\ &= \langle a^{\mathcal{A}}, l^{\mathcal{A}} \rangle_{\mathbf{MR}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}}) \\ &= \langle a^{\mathcal{B}}, l^{\mathcal{B}} \rangle_{\mathbf{MR}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}}) \\ &= \langle a^{\mathcal{B}}, l^{\mathcal{B}} \rangle_{\mathbf{MR}} \end{aligned}$$

Literals of the form $r_1 = r_2 \cup_{\mathbf{MR}} r_3$:

$$\begin{aligned} r_1^{\mathcal{B}} &= r_1^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}}) \\ &= \left(r_2^{\mathcal{A}} \cup_{\mathbf{MR}} r_3^{\mathcal{A}}\right) \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}}) \\ &= \left(r_2^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}})\right) \cup_{\mathbf{MR}} \left(r_3^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}})\right) \\ &= r_2^{\mathcal{B}} \cup_{\mathbf{MR}} r_3^{\mathcal{B}} \end{aligned}$$

Literals of the form $r_1 = r_2 -_{\mathbf{MR}} r_3$:

$$\begin{aligned} r_1^{\mathcal{B}} &= r_1^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{sLevelK}) \\ &= \left(r_2^{\mathcal{A}} -_{\mathbf{MR}} r_3^{\mathcal{A}}\right) \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{sLevelK}) \\ &= \left(r_2^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{sLevelK})\right) -_{\mathbf{MR}} \left(r_3^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{sLevelK})\right) \\ &= r_2^{\mathcal{B}} -_{\mathbf{MR}} r_3^{\mathcal{B}} \end{aligned}$$

Literals of the form $a = x[t]_{\mathbf{A}}$:

$$\begin{aligned} x[t]_{\mathbf{A}}^{\mathcal{B}} &= x^{\mathcal{B}}(t^{\mathcal{B}}) \\ &= x^{\mathcal{A}}(t^{\mathcal{B}}) \\ &= x^{\mathcal{A}}(t^{\mathcal{A}}) \\ &= x[t]_{\mathbf{A}}^{\mathcal{A}} \\ &= a^{\mathcal{A}} \\ &= a^{\mathcal{B}} \end{aligned}$$

Literals of the form $x_{new} = x\{t \leftarrow a\}_{\mathbf{A}}$: In this case, since ${x_{new}}^{\mathcal{A}}(t^{\mathcal{A}}) = a^{\mathcal{A}}$, it follows that ${x_{new}}^{\mathcal{B}}(t^{\mathcal{B}}) = a^{\mathcal{B}}$. Let us now consider an arbitrary $i \in \mathcal{B}_{\mathsf{thid}}$ such that $i \neq t^{\mathcal{A}}$. Then, ${x_{new}}^{\mathcal{A}}(i) = x^{\mathcal{A}}(i) = x^{\mathcal{B}}(i)$.

Literals of the form $l = z[t]_{\mathrm{L}}$:

$$\begin{aligned} z[t]_{\mathrm{L}}^{\mathcal{B}} &= z^{\mathcal{B}}(t^{\mathcal{B}}) \\ &= z^{\mathcal{A}}(t^{\mathcal{B}}) \\ &= z^{\mathcal{A}}(t^{\mathcal{A}}) \\ &= z[t]_{\mathrm{L}}^{\mathcal{A}} \\ &= l^{\mathcal{A}} \\ &= l^{\mathcal{B}} \end{aligned}$$

Literals of the form $z_{new} = z\{t \leftarrow l\}_{\mathrm{L}}$: In this case, since ${z_{new}}^{\mathcal{A}}(t^{\mathcal{A}}) = l^{\mathcal{A}}$, it follows that ${z_{new}}^{\mathcal{B}}(t^{\mathcal{B}}) = l^{\mathcal{B}}$. Let us now consider an arbitrary $i \in \mathcal{B}_{\mathsf{thid}}$ such that $i \neq t^{\mathcal{A}}$. Then, ${z_{new}}^{\mathcal{A}}(i) = z^{\mathcal{A}}(i) = z^{\mathcal{B}}(i)$.

Literals of the form $s = addr2set_{\mathsf{K}}(m, a, l)$: Let $x = a^{\mathcal{B}} = a^{\mathcal{A}}$. Then,

$$\begin{aligned} s^{\mathcal{B}} &= s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} \\ &= \left\{y \in \mathcal{A}_{\mathsf{addr}} \mid \exists\, p \in \mathcal{A}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{A}}, x, y, l, p) \in {reach_{\mathsf{K}}}^{\mathcal{A}}\right\} \cap \mathcal{B}_{\mathsf{addr}} \\ &= \left\{y \in \mathcal{B}_{\mathsf{addr}} \mid \exists\, p \in \mathcal{A}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{A}}, x, y, l, p) \in {reach_{\mathsf{K}}}^{\mathcal{A}}\right\} \\ &= \left\{y \in \mathcal{B}_{\mathsf{addr}} \mid \exists\, p \in \mathcal{B}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{B}}, x, y, l, p) \in {reach_{\mathsf{K}}}^{\mathcal{B}}\right\} \end{aligned}$$

It just remains to see that the last equality holds. Let

- $S_{\mathcal{B}} = \left\{y \in \mathcal{B}_{\mathsf{addr}} \mid \exists\, p \in \mathcal{B}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{B}}, x, y, l, p) \in {reach_{\mathsf{K}}}^{\mathcal{B}}\right\}$, and
- $S_{\mathcal{A}} = \left\{y \in \mathcal{B}_{\mathsf{addr}} \mid \exists\, p \in \mathcal{A}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{A}}, x, y, l, p) \in {reach_{\mathsf{K}}}^{\mathcal{A}}\right\}$

We first show that $S_{\mathcal{A}} \subseteq S_{\mathcal{B}}$. Let $y \in S_{\mathcal{A}}$. Then it exists $p \in \mathcal{A}_{\mathsf{path}}$ such that $(m^{\mathcal{A}}, x, y, l, p) \in {reach_{\mathsf{K}}}^{\mathcal{A}}$. Then, by definition of $reach_{\mathsf{K}}$ there are two possible cases.

- If $p = \epsilon$ and $x = y$, then $(m^{\mathcal{B}}, x, y, l, \epsilon^{\mathcal{B}}) \in {reach_{\mathsf{K}}}^{\mathcal{B}}$ and therefore $y \in S_{\mathcal{B}}$.
- Otherwise, there exists $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ s.t.,

$$\begin{array}{llll} i) & p = [a_1, \ldots, a_n] & iii) & m^{\mathcal{A}}(a_r).next^{\mathcal{A}}(l) = a_{r+1}, \text{for } 1 \leq r < n \\ ii) & x = a_1 & iv) & m^{\mathcal{A}}(a_n).next^{\mathcal{A}}(l) = y \end{array}$$

Then, we only need to find $\tilde{a}_1, \ldots, \tilde{a}_m \in \mathcal{B}_{\mathsf{addr}}$ s.t.,

$$\begin{array}{llll} i) & q = [\tilde{a}_1, \ldots, \tilde{a}_m] & iii) & m^{\mathcal{B}}(\tilde{a}_r).next^{\mathcal{B}}(l) = \tilde{a}_{r+1}, \text{for } 1 \leq r < m \\ ii) & x = \tilde{a}_1 & iv) & m^{\mathcal{B}}(\tilde{a}_m).next^{\mathcal{B}}(l) = y \end{array}$$

We define $\tilde{a}_1 = a_1 = x$ and $\tilde{a}_2 = first_{\mathsf{K}}(m^{\mathcal{A}}, \tilde{a}_1, l, \mathcal{B}_{\mathsf{addr}})$. Then, $\tilde{a}_2 = m^{\mathcal{B}}(\tilde{a}_1).next^{\mathcal{B}}(l)$ and $\tilde{a}_2 \in \mathcal{B}_{\mathsf{addr}}$ and thus $\tilde{a}_2 \in \mathcal{B}_{\mathsf{addr}}$. If $\tilde{a}_2 = y$ there is nothing else to prove. On the other hand, if $\tilde{a}_2 \neq y$ then we proceed in the same way to define $\tilde{a}_3$ and so on until $\tilde{a}_{m+1} = y$. Notice that this way, y is guaranteed to be found in at most n steps.
To show that $S_{\mathcal{B}} \subseteq S_{\mathcal{A}}$ we proceed in a similar way. Let $y \in S_{\mathcal{B}}$. Then $x = y$ and $p = \epsilon$ and thus $(m^{\mathcal{A}}, x, y, l, \epsilon^{\mathcal{A}}) \in {reach_{\mathsf{K}}}^{\mathcal{A}}$, or exists $a_1, \ldots, a_n \in \mathcal{B}_{\mathsf{addr}}$ such that

$$\begin{array}{llll} i) & p = [a_1, \ldots, a_n] & iii) & m^{\mathcal{B}}(a_r).next^{\mathcal{B}}(l) = a_{r+1}, \text{for } 1 \leq r < n \\ ii) & x = a_1 & iv) & m^{\mathcal{B}}(a_n).next^{\mathcal{B}}(l) = y \end{array}$$

As we know that $a_1, \ldots, a_n, y \in \mathcal{B}_{\mathsf{addr}}$, by definition of $first_{\mathsf{K}}$ we know that there exists $s \geq 1$ s.t.,

$$m^{\mathcal{A}}\Big(\cdots\Big(m^{\mathcal{A}}\,(a_1)\,.\underbrace{next^{\mathcal{A}}\,(l)\Big)\cdots\Big).next^{\mathcal{A}}\,(l)}_{s} = a_2$$

Let then $a_1^1, \ldots, a_1^{s-1} \in \mathcal{A}_{\mathsf{addr}}$ such that

$$\begin{aligned} m^{\mathcal{A}}(a_1).next^{\mathcal{A}}(l) &= a_1^1 \\ m^{\mathcal{A}}(a_1^1).next^{\mathcal{A}}(l) &= a_1^2 \\ &\vdots \\ m^{\mathcal{A}}(a_1^{s-1}).next^{\mathcal{A}}(l) &= a_2 \end{aligned}$$

We then use $a_1^1, \ldots, a_1^{s-1}$ to construct the section of a path q that goes from a_1 up to a_2. Finally we use the same approach to finish the construction of such a path in $\mathcal{A}$. Then we have that $(m^{\mathcal{A}}, x, y, l, q^{\mathcal{A}}) \in reach_{\mathsf{K}}{}^{\mathcal{A}}$. Hence, $y \in S_{\mathcal{A}}$.

Literals of the form $p = getp_{\mathsf{K}}(m, a, b, l)$**:** We consider two possible cases.

- Case $b^{\mathcal{A}} \in addr2set_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, l)$.
  Since $(m^{\mathcal{A}}, a^{\mathcal{A}}, b^{\mathcal{A}}, l, p^{\mathcal{A}}) \in reach_{\mathsf{K}}{}^{\mathcal{A}}$, it is enough to prove:

  $$\text{if} \quad (m^{\mathcal{A}}, x, y, l, q) \in reach_{\mathsf{K}}{}^{\mathcal{A}} \quad \text{then} \quad (m^{\mathcal{B}}, x, y, l, compress(q, \mathcal{B}_{\mathsf{addr}})) \in reach_{\mathsf{K}}{}^{\mathcal{B}}$$

  for each $x, y \in \mathcal{B}_{\mathsf{addr}}$ and $q \in \mathcal{A}_{\mathsf{path}}$. If $(m^{\mathcal{A}}, x, y, l, q) \in reach_{\mathsf{K}}{}^{\mathcal{A}}$, $x = y$ and $q = \epsilon$, then $(m^{\mathcal{B}}, x, y, l, compress(q, \mathcal{B}_{\mathsf{addr}})) \in reach_{\mathsf{K}}{}^{\mathcal{B}}$. Otherwise, there exist $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that:

i)	$q = [a_1, \ldots, a_n]$	*iii)*	$m^{\mathcal{A}}(a_r).next^{\mathcal{A}}(l) = a_{r+1}$, for $1 \leq r < n$
ii)	$x = a_1$	*iv)*	$m^{\mathcal{A}}(a_n).next^{\mathcal{A}}(l) = y$

  Then, we proceed by induction on n.
  - If $n = 1$, then $q = [a_1]$ and therefore $compress(q, \mathcal{B}_{\mathsf{addr}}) = [a_1]$, as $x = a_1 \in \mathcal{B}_{\mathsf{addr}}$. Besides, $m^{\mathcal{A}}(a_1).next^{\mathcal{A}}(l) = y$ which implies that $m^{\mathcal{B}}(a_1).next^{\mathcal{B}}(l) = y$. Therefore, $(m^{\mathcal{B}}, x, y, l, compress(q, \mathcal{B}_{\mathsf{addr}})) \in reach_{\mathsf{K}}{}^{\mathcal{B}}$.
  - If $n > 1$, then let $a_i = first_{\mathsf{K}}(m^{\mathcal{A}}, x, l, \mathcal{B}_{\mathsf{addr}})$. As

    $$q = [x = a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n]$$

    we have that

    $$compress(q, \mathcal{B}_{\mathsf{addr}}) = [x = a_1] \circ compress([a_i, a_{i+1}, \ldots, a_n], \mathcal{B}_{\mathsf{addr}})$$

    Besides, as $(m^{\mathcal{A}}, a_i, y, l, [a_i, a_{i+1}, \ldots, a_n]) \in reach_{\mathsf{K}}{}^{\mathcal{A}}$, by induction we have that

    $$(m^{\mathcal{B}}, a_i, y, l, compress([a_i, a_{i+1}, \ldots, a_n], \mathcal{B}_{\mathsf{addr}})) \in reach_{\mathsf{K}}{}^{\mathcal{B}}$$

    Moreover $m^{\mathcal{B}}(x).next^{\mathcal{B}}(l) = a_i$ and therefore

    $$(m^{\mathcal{B}}, x, y, l, compress(q, \mathcal{B}_{\mathsf{addr}})) \in reach_{\mathsf{K}}{}^{\mathcal{B}}$$

- Case $b^{\mathcal{A}} \notin addr2set_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, l)$.
  In such case we have that $p^{\mathcal{A}} = \epsilon$, which implies that $p^{\mathcal{B}} = \epsilon$. Then using a reasoning similar to the previous case we can deduce that $b^{\mathcal{B}} \notin addr2set_{\mathsf{K}}(m^{\mathcal{B}}, a^{\mathcal{B}}, l)$.

Literals of the form $a = fstlock_{\mathbf{K}}(m, p, l)$**:** Consider the case $p = \epsilon$: then $fstlock_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, \epsilon^{\mathcal{A}}, l^{\mathcal{A}}) = null^{\mathcal{A}}$. At the same time, we know that $\epsilon^{\mathcal{B}} = compress(\epsilon^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$ and so $fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}, l^{\mathcal{B}}) = null^{\mathcal{B}}$. Let's now consider the case at which $p = [a_1, \ldots, a_n]$. There are two scenarios to consider.

- If for all $1 \leq k \leq n$, $m^{\mathcal{A}}(a_k^{\mathcal{A}}).lockid(l) = \oslash$, then we have that

$$fstlock_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}, l^{\mathcal{A}}) = null^{\mathcal{A}}$$

  Notice that function *compress* returns a subset of the path it receives with the property that all addresses in the returned path belong to the received set. Hence, if $[\tilde{a}_1, \ldots, \tilde{a}_m] = p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$, we know that $\{\tilde{a}_1, \ldots, \tilde{a}_m\} \subseteq \mathcal{B}_{\mathsf{addr}}$ and therefore for all $1 \leq j \leq m$, $m^{\mathcal{B}}(\tilde{a}_j).lockid^{\mathcal{B}}(l^{\mathcal{B}}) = \oslash$. Then, we can conclude that in fact $fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, p^{\mathcal{B}}, l^{\mathcal{B}}) = null^{\mathcal{B}}$.
- If there is a k, with $1 \leq k \leq n$, such that for all $1 \leq j < k$, $m^{\mathcal{A}}(a_j^{\mathcal{A}}).lockid(l) = \oslash$ and $m^{\mathcal{A}}(a_k^{\mathcal{A}}).lockid(l) \neq \oslash$ then since $a^{\mathcal{B}} = a^{\mathcal{A}}$, we can say that $a^{\mathcal{B}} = a^{\mathcal{A}} = x \in \mathcal{B}_{\mathsf{addr}}$. It then remains to verify whether

$$\text{if} \quad x = fstlock_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}, l^{\mathcal{A}}) \quad \text{then} \quad x = fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}), l^{\mathcal{B}})$$

  By definition of $fstlock_{\mathsf{K}}$ we have that $x = a_k^{\mathcal{A}}$ and by construction of $\mathcal{B}_{\mathsf{addr}}$ we know that $a_k^{\mathcal{A}} \in \mathcal{B}_{\mathsf{addr}}$. Let $[\tilde{a}_1, \ldots, \tilde{a}_i, \ldots, \tilde{a}_m] = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$ such that $\tilde{a}_i = a_k^{\mathcal{A}}$. We had that $\tilde{a}_j \in \mathcal{B}_{\mathsf{addr}}$ for all $1 \leq j \leq m$. As *compress* preserves the order and for all $1 \leq j < k$, $m^{\mathcal{A}}(a_j^{\mathcal{A}}).lockid^{\mathcal{A}}(l^{\mathcal{A}}) = \oslash$, we have that for all $1 \leq j < i$, $m^{\mathcal{B}}(\tilde{a}_j).lockid^{\mathcal{B}}(l^{\mathcal{B}}) = \oslash$. Besides $m^{\mathcal{B}}(\tilde{a}_i).lockid^{\mathcal{B}}(l^{\mathcal{B}}) \neq \oslash$. Then:

$$\begin{aligned} fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, compress(p^{\mathcal{A}}), l^{\mathcal{B}}) &= fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, [\tilde{a}_1, \ldots, \tilde{a}_m], l^{\mathcal{B}}) \\ &= \tilde{a}_i \\ &= a_k^{\mathcal{A}} \\ &= x \end{aligned}$$

Literals of the form $ordList(m, p)$**:** Assume that $(m^{\mathcal{A}}, p^{\mathcal{A}}) \in ordList^{\mathcal{A}}$. We would like to verify whether $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordList^{\mathcal{B}}$ i.e., $(m^{\mathcal{B}}, compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordList^{\mathcal{B}}$. We proceed by induction on p.

- If $p = \epsilon$, by definition of *compress* and *ordList*, we have that $(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}) \in ordList^{\mathcal{B}}$.
- If $p = [a_1]$, we know that $(m^{\mathcal{A}}, [a_1]^{\mathcal{A}}) \in ordList^{\mathcal{A}}$ and that $p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$. Then, if $a_1^{\mathcal{A}} \in \mathcal{B}_{\mathsf{addr}}$, we have that $p^{\mathcal{B}} = [a_1]^{\mathcal{B}}$ and then clearly $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordList^{\mathcal{B}}$ holds. On the other hand, if $a_1^{\mathcal{A}} \notin \mathcal{B}_{\mathsf{addr}}$, then $p^{\mathcal{B}} = \epsilon^{\mathcal{B}}$ and once more $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordList^{\mathcal{B}}$ holds.
- If $p = [a_1, \ldots, a_{n+1}]$ with $n \geq 1$, then we have two possible cases to bear in mind. If we consider the case at which $a_1^{\mathcal{A}} \notin \mathcal{B}_{\mathsf{addr}}$ then

$$compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) = compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$$

  and, as by induction we have that $(m^{\mathcal{B}}, compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordList^{\mathcal{B}}$, we conclude that $(m^{\mathcal{B}}, compress([a_1, a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordList^{\mathcal{B}}$. On the other hand, if we assume $a_1^{\mathcal{A}} \in \mathcal{B}_{\mathsf{addr}}$, by induction, $(m^{\mathcal{B}}, compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordList^{\mathcal{B}}$. Besides, as $m^{\mathcal{A}}(a_1^{\mathcal{A}}).key^{\mathcal{A}} \preceq m^{\mathcal{A}}(a_2^{\mathcal{A}}).key^{\mathcal{A}}$ we can deduce that $m^{\mathcal{B}}(a_1^{\mathcal{A}}).key^{\mathcal{B}} \preceq m^{\mathcal{B}}(a_2^{\mathcal{A}}).key^{\mathcal{B}}$. And so, $(m^{\mathcal{B}}, compress([a_1, a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordList^{\mathcal{B}}$.

Literals of the form $\neg ordList(m, p)$**:** Let's assume that $(m^{\mathcal{A}}, p^{\mathcal{A}}) \notin ordList^{\mathcal{A}}$. We want to see that $(m^{\mathcal{B}}, p^{\mathcal{B}}) \notin ordList^{\mathcal{B}}$. If $(m^{\mathcal{A}}, p^{\mathcal{A}}) \notin ordList^{\mathcal{A}}$, then it means that $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and $m^{\mathcal{A}}(a_{i+1}).key^{\mathcal{A}} \preceq m^{\mathcal{A}}(a_i).key^{\mathcal{A}}$ and $m^{\mathcal{A}}(a_{i+1}).key^{\mathcal{A}} \neq m^{\mathcal{A}}(a_i).key^{\mathcal{A}}$ for some $i \in 1, \ldots, n-1$. Let that i be the one such that for all $j < i$, $m^{\mathcal{A}}(a_j).key^{\mathcal{A}} \preceq m^{\mathcal{A}}(a_{j+1}).key^{\mathcal{A}}$. Then, by Lemma 5.3 we have $unordered(m^{\mathcal{A}}, [a_1, \ldots, a_n]^{\mathcal{A}}) \neq \emptyset$ and besides $\{a_i^{\mathcal{A}}, a_{i+1}^{\mathcal{A}}\} \subseteq unordered(m^{\mathcal{A}}, [a_1, \ldots, a_n]^{\mathcal{A}}) \subseteq \mathcal{B}_{\mathsf{addr}}$. This means that

$$compress([a_1, \ldots, a_n]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) = [\tilde{a}_1, \ldots, a_i, a_{i+1}, \ldots, \tilde{a}_m]^{\mathcal{B}}$$

Therefore, since $m^{\mathcal{B}}(a_{i+1}).key^{\mathcal{B}} \preceq m^{\mathcal{B}}(a_i).key^{\mathcal{B}}$ and $m^{\mathcal{B}}(a_{i+1}).key^{\mathcal{B}} \neq m^{\mathcal{B}}(a_i).key^{\mathcal{B}}$, we have that

$$(m^{\mathcal{B}}, compress([a_1, \ldots, a_n]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \notin ordList^{\mathcal{B}}. \qquad \square$$

5.2 A Combination-based Decision Procedure for TSL_K

Lemma 5.4 enables a brute force method to automatically check whether a set of normalized TSL_K-literals is satisfiable. However, such a method is not efficient in practice. We describe now how to obtain a more efficient decision procedure for TSL_K applying a many-sorted variant [TZ04] of the Nelson-Oppen combination method [NO79], by combining the decision procedures for the underlying theories. This combination method requires that the theories fulfill some conditions. First, each theory must have a decision procedure. Second, two theories can only share sorts (but not functions or predicates). Third, when two theories are combined, either both theories are stable infinite or one of them is polite with respect to the underlying sorts that it shares with the other. The stable infinite condition for a theory establishes that if a formula has a model then it has a model with infinite cardinality. In our case, some theories are not stable infinite. For example, $T_{\mathsf{level}_\mathsf{K}}$ is not stably infinite, T_{ord}, and T_{thid} need not be stable infinite in some instances. The observation that the condition of stable infinity may be cumbersome in the combination of theories for data structures was already made in [RZ06a] where they suggest the condition of *politeness*:

Definition 5.3 (Smoothness).
Let $\Sigma = (S, F, P)$ be a signature, $S_0 = \{\sigma_1, \ldots, \sigma_n\} \subseteq \mathcal{S}$ be a set of sorts, and T be a Σ-theory. T is smooth with respect to S_0 if:

(i) for every T-satisfiable quantifier-free Σ-formula φ,

(ii) for every T-interpretation $\mathcal{A}$ satisfying φ,

(iii) for every cardinal number $k_1, \ldots, k_n$ such that $k_i \geq |\mathcal{A}_{\sigma_i}|$

there exists a T-interpretation $\mathcal{B}$ satisfying φ such that $|\mathcal{B}_{\sigma_i}| = k_i$, for $i = 1, \ldots, n$. †

Definition 5.4 (Finite witnessability).
Let $\Sigma = (S, F, P)$ be a signature, $S_0 \subseteq S$ be a set of sorts, ant T be a Σ-theory. T is finitely witnessable with respect to S_0 if there exists a computable function witness that for every quantifier-free Σ-formula φ returns a quantifier-fee Σ-formula $\psi = witness(\varphi)$ such that

(i) φ and $(\exists \overline{v})\psi$ are T-equivalent, where $\overline{v} = vars(\psi) \setminus vars(\varphi)$, and

(ii) if ψ is T-satisfiable then there exists a T-interpretation $\mathcal{A}$ satisfying ψ such that the domain $\mathcal{A}_\sigma$ interpreting the sort σ in $\mathcal{A}$ is the (finite) set $[vars_\sigma(\psi)]^{\mathcal{A}}$ of elements in $\mathcal{A}$ interpreting the variables of sort σ in ψ (in symbols, $\mathcal{A}_\sigma = [vars_\sigma(\psi)]^{\mathcal{A}}$), for each $\sigma \in S_0$. †

Definition 5.5 (Politeness).
Let $\Sigma = (S, F, P)$ be a signature, $S_0 \subseteq S$ be a set of sorts and T be a Σ-theory. T is polite with respect to S_0 if it is both smooth and finitely witnessable with respect to S_0. †

Smoothness guarantees that interpretations can be enlarged as needed. Finite witnessability gives a procedure to produce a model in which every element is represented by a variable. The Finite Model Property, Lemma 5.4 above, guarantees that every sub-theory of $\mathsf{TSL_K}$ is finite witnessable since one can add as many fresh variables as the bound for the corresponding sort in the lemma, even one for each element in the domain of the sorts. The smoothness property can be shown for:

$$T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{path}} \cup T_{\mathsf{set}} \cup T_{\mathsf{settid}} \cup T_{\mathsf{mrgn}} \cup T_{\mathsf{aarr}} \cup T_{\mathsf{larr}}$$

with respect to sorts addr, $\mathsf{level_K}$, elem, ord and thid. Moreover, these theories can be combined because all of them are stably infinite. The following can also be combined:

$$T_{\mathsf{level_K}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}}$$

because they do not share any sorts, so combination is trivial. The many-sorted Nelson-Oppen method allows to combine the first collection of theories with the second. Regarding the decision procedures for each individual theory, $T_{\mathsf{level_K}}$ is trivial since it is just a finite set of naturals with order. For T_{ord} we can adapt a decision procedure for dense orders as the reals [Tar51], or other appropriate theory. For instance, if no dense order is required, we can consider Presburger Arithmetic equiped only with 0, *succ* and $<$. For T_{cell} we can use a decision procedure for recursive data structures [Opp80]. T_{mem} is the theory of arrays [ARR03]. T_{set}, T_{settid} and T_{mrgn} are theories of (finite) sets for which there are many decision procedures [Zar03, KNR05]. For theories T_{aarr} and T_{larr} we can use a decision procedure for arrays [BMS06]. The remaining theories are T_{reach} and T_{bridge}. Following the approaches in [RZ06a, SS10] we extend a decision procedure for the theory T_{path} of finite sequences of (non-repeated) addresses with the auxiliary functions and predicates shown in Fig. 5.5, and combine this theory with the theories that are part of $\mathsf{TSL_K}$ (except from T_{reach}) to obtain:

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{path}} \cup T_{\mathsf{set}} \cup T_{\mathsf{settid}} \cup T_{\mathsf{mrgn}} \cup T_{\mathsf{aarr}} \cup T_{\mathsf{larr}}$$

Using T_{path} all symbols in T_{reach} can be easily defined. Once more, we pick the theory of finite sequences of addresses defined by $T_{\mathsf{fseq}} = \left(\Sigma_{\mathsf{fseq}}, \mathsf{TGen}\right)$, where:

$$\begin{array}{rcl}
\Sigma_{\mathsf{fseq}} & = & \Big(\{ \quad \mathsf{addr}, \mathsf{fseq} \qquad\qquad\qquad\qquad\quad \}, \\
& & \quad \{ \quad \mathit{nil} \quad : \quad \mathsf{fseq}, \\
& & \qquad \mathit{cons} \quad : \quad \mathsf{addr} \times \mathsf{fseq} \rightarrow \mathsf{fseq}, \\
& & \qquad \mathit{hd} \quad : \quad \mathsf{fseq} \rightarrow \mathsf{addr}, \\
& & \qquad \mathit{tl} \quad : \quad \mathsf{fseq} \rightarrow \mathsf{fseq} \qquad\quad \}, \\
& & \{ \qquad\qquad\qquad\qquad\qquad\qquad\quad \} \Big)
\end{array}$$

and TGen as the class of term-generated structures that satisfy the axioms of distinctness, uniqueness and generation of sequences using constructors, as well as acyclicity (see, for example [BM07]). Let Σ_{path} be Σ_{fseq} extended with the symbols of Fig. 5.5 and let *PATH* be the set of axioms of T_{fseq} including the ones in Fig. 5.5. Then, we can formally define $T_{\mathsf{path}} = \left(\Sigma_{\mathsf{path}}, \mathsf{ETGen}\right)$ where

$$\mathsf{ETGen} = \left\{\mathcal{A}^{\Sigma_{\mathsf{path}}} \mid \mathcal{A}^{\Sigma_{\mathsf{path}}} \vDash \mathit{PATH} \text{ and } \mathcal{A}^{\Sigma_{\mathsf{fseq}}} \in \mathsf{TGen}\right\}$$

Next, we extend T_{SLKBase} with definitions for translating all missing functions and predicates from Σ_{reach} and Σ_{bridge} appearing in normalized $\mathsf{TSL_K}$-literals by definitions from T_{SLKBase}. Let *GAP* be the set

of axioms that define ϵ, [_], *append*, $reach_{\mathsf{K}}$, *path2set*, $getp_{\mathsf{K}}$, $fstlock_{\mathsf{K}}$ and *ordList*. Some of the equivalences for *GAP* were given in Section 4.2. Hence, here we limit ourselves only to the new equivalences. For instance:

$$\begin{array}{rcl} isreachp_{\mathsf{K}}(m,a_1,a_2,l,p) & \rightarrow & getp_{\mathsf{K}}(m,a_1,a_2,l) = p \\ \neg isreachp(m,a_1,a_2,l,p) & \rightarrow & getp_{\mathsf{K}}(m,a_1,a_2,l) = nil \\ ispath\,(p) \wedge fstmark\,(m,p,l,i) & \leftrightarrow & fstlock_{\mathsf{K}}\,(m,p,l) = i \\ ispath\,(p) \wedge ordPath\,(m,p) & \leftrightarrow & ordList\,(m,p) \end{array}$$

We now define $\widehat{\mathsf{TSL_K}} = (\Sigma_{\widehat{\mathsf{TSL_K}}}, \widehat{\mathsf{ETGen}})$ where

$$\Sigma_{\widehat{\mathsf{TSL_K}}} = \Sigma_{T_{\mathsf{SLKBase}}} \cup \{append, reach_{\mathsf{K}}, path2set, getp_{\mathsf{K}}, fstlock_{\mathsf{K}}, ordList\}$$

and $\widehat{\mathsf{ETGen}} := \left\{\mathcal{A}^{\Sigma_{\widehat{\mathsf{TSL_K}}}} \mid \mathcal{A}^{\Sigma_{\widehat{\mathsf{TSL_K}}}} \models GAP \text{ and } \mathcal{A}^{\Sigma_{T_{\mathsf{SLKBase}}}} \in \mathsf{ETGen}\right\}$.

Using the definitions of *GAP* it is easy to prove that if Γ is a set of normalized $\mathsf{TSL_K}$-literals, then Γ is $\mathsf{TSL_K}$-satisfiable iff Γ is $\widehat{\mathsf{TSL_K}}$-satisfiable. Therefore, $\widehat{\mathsf{TSL_K}}$ can be used in place of $\mathsf{TSL_K}$ for satisfiability checking. The reduction from $\widehat{\mathsf{TSL_K}}$ into T_{SLKBase} is performed in two steps. First, by the finite model theorem (Lemma 5.4), it is always possible to calculate an upper bound in the number of elements of

$app : \mathsf{fseq} \times \mathsf{fseq} \rightarrow \mathsf{fseq}$
$app(nil, l) = l$
$app(cons(a, l), l') = cons(a, app(l, l'))$

$fseq2set : \mathsf{fseq} \rightarrow \mathsf{set}$
$fseq2set(nil) = \emptyset$
$fseq2set(cons(a, l)) = \{a\} \cup fseq2set(l)$

$ispath : \mathsf{fseq}$
$ispath(nil)$
$ispath(cons(a, nil))$
$\{a\} \not\subseteq fseq2set(l) \wedge ispath(l) \rightarrow ispath(cons(a, l))$

$last : \mathsf{fseq} \rightarrow \mathsf{addr}$
$last(cons(a, nil)) = a$
$l \neq nil \rightarrow last(cons(a, l)) = last(l)$

$isreach_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K}$
$isreach_{\mathsf{K}}(m, a, a, l)$
$m[a].next[l] = a' \wedge isreach_{\mathsf{K}}(m, a', b, l) \rightarrow isreach_{\mathsf{K}}(m, a, b, l)$

$isreachp_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \times \mathsf{fseq}$
$isreachp_{\mathsf{K}}(m, a, a, l, nil)$
$m[a].next[l] = a' \wedge isreachp(m, a', b, l, p) \rightarrow isreachp(m, a, b, l, cons(a, p))$

$fstmark : \mathsf{mem} \times \mathsf{fseq} \times \mathsf{level_K} \times \mathsf{addr}$
$fstmark(m, nil, l, null)$
$p \neq nil \wedge p = cons(a, q) \wedge m[a].lockid[l] \neq \oslash \rightarrow fstmark(m, p, l, a)$
$p \neq nil \wedge p = cons(a, q) \wedge m[a].lockid[l] = \oslash \wedge fstmark(m, q, l, b) \rightarrow fstmark(m, p, l, b)$

$ordPath : \mathsf{mem} \times \mathsf{fseq}$
$ordPath(h, nil)$
$ordPath(h, cons(a, nil))$
$h[a].next[0] = a' \wedge h[a].key \preceq h[a'].key \wedge p = cons(a, q) \wedge ordPath(h, q) \rightarrow ordPath(h, p)$

Figure 5.5: Functions, predicates and axioms of T_{path}

sort addr, elem, thid, ord and level in a model (if there is one model), based on the input formula. Therefore, one can introduce one variable per element of each of these sorts and unfold all definitions in *PATH* and *GAP*, by symbolic expansion, leading to terms in Σ_{fseq}, and thus, in T_{SLKBase}. This way, it is always possible to reduce a $\widehat{\mathsf{TSL}_\mathsf{K}}$-satisfiability problem of normalized literals into a T_{SLKBase}-satisfiability problem. Hence, using a decision procedure for T_{SLKBase} we obtain a decision procedure for $\widehat{\mathsf{TSL}_\mathsf{K}}$, and thus, for TSL_K. Notice, for instance, that the predicate *subPath* : path $\times$ path for ordered lists can be defined using only *path2set* as:

$$subPath(p_1, p_2) \mathrel{\hat{=}} path2set(p_1) \subseteq path2set(p_2)$$

Example 5.2
We now illustrate how reduction from $\widehat{\mathsf{TSL}_\mathsf{K}}$ to T_{SLKBase} is performed. For the sake of simplicity, let just consider the formula $\varphi = SkipList_4(h, sl.head, sl.tail)$. According to the definition we gave in Example 3.3,

$$\begin{aligned}
\varphi \quad \hat{=} \quad & OList(h, sl.head, 0) \ \wedge && (5.1)\\
& \begin{pmatrix} sl.tail.next[0] = null \wedge sl.tail.next[1] = null \\ sl.tail.next[2] = null \wedge sl.tail.next[3] = null \end{pmatrix} \wedge && (5.2)\\
& \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} && (5.3)
\end{aligned}$$

For simplicty, from now on, we use *head* for *sl.head* and *tail* for *sl.tail*. Notice that φ is not in TSL_K. In fact, it contains some functions like *OList* and *SubList* which do not belong to TSL_K but give us the intuition of what a skiplist should look like. Hence, we first rewrite φ using terms of TSL_K:

$$\begin{aligned}
(5.1)\quad & \overbrace{p = getp_\mathsf{K}(h, head, null, 0)}^{\Sigma_{\mathsf{bridge}}} \wedge \overbrace{ordList(h, p)}^{\Sigma_{\mathsf{bridge}}} \wedge \\
(5.2)\quad & \overbrace{c_{tail} = h[tail]}^{\Sigma_{\mathsf{mem}}} \wedge \overbrace{a_{tail_0} = c_{tail}.next[0]}^{\Sigma_{\mathsf{cell}}} \wedge \overbrace{a_{tail_0} = null}^{\Sigma_{\mathsf{addr}}} \\
& \qquad\qquad \wedge \quad \vdots \quad \wedge \quad \vdots \\
& \qquad\qquad \wedge \overbrace{a_{tail_3} = c_{tail}.next[3]}^{\Sigma_{\mathsf{cell}}} \wedge \overbrace{a_{tail_3} = null}^{\Sigma_{\mathsf{addr}}} \wedge \\
(5.3)\quad & \overbrace{p_{tail_0} = getp_\mathsf{K}(h, head, tail, 0)}^{\Sigma_{\mathsf{bridge}}} \wedge \quad \cdots \quad \wedge \overbrace{p_{tail_3} = getp_\mathsf{K}(h, head, tail, 3)}^{\Sigma_{\mathsf{bridge}}} \wedge \\
& \overbrace{s_{tail_0} = path2set(p_{tail_0})}^{\Sigma_{\mathsf{bridge}}} \wedge \quad \cdots \quad \wedge \overbrace{s_{tail_3} = path2set(p_{tail_3})}^{\Sigma_{\mathsf{bridge}}} \wedge \\
& \overbrace{s_{tail_1} \subseteq s_{tail_0}}^{\Sigma_{\mathsf{set}}} \wedge \quad \cdots \quad \wedge \overbrace{s_{tail_3} \subseteq s_{tail_2}}^{\Sigma_{\mathsf{set}}}
\end{aligned}$$

Notice that after been rewriten, we already have a conjuntion of literals belonging to $\widehat{\mathsf{TSL}_\mathsf{K}}$. We can go a little further, and rewrite φ using normalized literals. Some of the literals are already normalized, like $ordList(h, p)$, while other must be rewriten, like $a_{tail_i} = c_{tail}.next[i]$ or $s_{tail_j} \subseteq s_{tail_{j-1}}$ to:

$$\begin{aligned}
& c_{tail} = mkcell(e_0, k_0, a_{tail_0}, a_0^1, a_0^2, a_0^3, t_0^0, t_0^1, t_0^2, t_0^3) \ \wedge \ a_{tail_0} = null \\
& c_{tail} = mkcell(e_1, k_1, a_1^0, a_{tail_1}, a_1^2, a_1^3, t_1^0, t_1^1, t_1^2, t_1^3) \ \wedge \ a_{tail_1} = null \\
& c_{tail} = mkcell(e_2, k_2, a_2^0, a_2^1, a_{tail_2}, a_2^3, t_2^0, t_2^1, t_2^2, t_2^3) \ \wedge \ a_{tail_2} = null \\
& c_{tail} = mkcell(e_3, k_3, a_3^0, a_3^1, a_3^2, a_{tail_3}, t_3^0, t_3^1, t_3^2, t_3^3) \ \wedge \ a_{tail_3} = null
\end{aligned}$$

and

$$s_{tail_0} = s_{tail_0} \cup s_{tail_1} \wedge s_{tail_1} = s_{tail_1} \cup s_{tail_2} \wedge s_{tail_2} = s_{tail_2} \cup s_{tail_3}$$

respectively. So far, we have

$$\begin{array}{rcl} V_{\mathsf{elem}}(\varphi) & = & \{e_0, e_1, e_2, e_3\} \\ V_{\mathsf{addr}}(\varphi) & = & \{head, tail\} \cup \{a_{tail_i} \mid 1 \in 0..3\} \cup (\{a_i^j \mid i, j \in 0..3 \wedge i \neq j\} \\ V_{\mathsf{mem}}(\varphi) & = & \{h\} \\ V_{\mathsf{path}}(\varphi) & = & \{p\} \cup \{p_{tail_i} \mid i \in 0..3\} \\ V_{\mathsf{thid}}(\varphi) & = & \emptyset \\ V_{\mathsf{ord}}(\varphi) & = & \emptyset \\ V_{\mathsf{aarr}}(\varphi) & = & \emptyset \end{array}$$

Therefore, assuming φ holds in a model $\mathcal{A}$, when constructing a finite model $\mathcal{B}$, we can ensure the following bounds for the domains of each sort in $\mathcal{B}$:

$$\begin{array}{rcl} |\mathcal{B}_{\mathsf{level_K}}| & \leq & \mathsf{K} = 4 \\ |\mathcal{B}_{\mathsf{thid}}| & \leq & \overline{t} + 1 + \mathsf{K}\,\overline{m}\,\overline{a} = 0 + 1 + 4 \times 1 \times 18 = 73 \\ |\mathcal{B}_{\mathsf{addr}}| & \leq & \overline{a} + 1 + \mathsf{K}\,\overline{m}\,\overline{a} + \overline{p}^2 + \overline{p}^3 + 2\,\overline{m}\,\overline{p} + \overline{x}\,|\mathcal{B}_{\mathsf{thid}}| \\ & = & 18 + 1 + 4 \times 1 \times 18 + 5^2 + 5^3 + 2 \times 1 \times 5 + 0 \times 73 = 251 \\ |\mathcal{B}_{\mathsf{elem}}| & \leq & \overline{e} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}| = 4 + 1 \times 251 = 255 \\ |\mathcal{B}_{\mathsf{ord}}| & \leq & \overline{o} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}| = 0 + 1 \times 251 = 251 \end{array}$$

In fact, when we construct the domains for $\mathcal{B}$, we find out that we only require the follwing sets, which clearly are under the bounds we have calculated:

$$\begin{array}{rcl} |\mathcal{B}_{\mathsf{level_K}}| & = & \{0, 1, 2, 3\} \\ |\mathcal{B}_{\mathsf{thid}}| & = & \{\oslash\} \\ |\mathcal{B}_{\mathsf{addr}}| & = & \{0\text{x}00, 0\text{x}01, 0\text{x}08\} \\ |\mathcal{B}_{\mathsf{elem}}| & = & \{-\infty, +\infty\} \\ |\mathcal{B}_{\mathsf{ord}}| & = & \{-\infty, +\infty\} \end{array}$$

Then, it just remains to reduce literals from Σ_{bridge} using the definitions of *GAP* and *PATH*, to end up with terms in T_{SLKBase}. This way, we get:

Σ_{bridge}-term	T_{SLKBase}-term
$p = getp_{\mathsf{K}}(h, head, tail, 0)$	$h[head].next[0] = tail \wedge h[tail].next[0] = null \wedge isreachp_{\mathsf{K}}(h, null, null, 0, nil)$
$ordList(h, p)$	$h[head].next[0] = tail \wedge h[head].key \preceq h[tail].key \wedge p = cons(head, cons(tail, nil)) \wedge ordPath(h, cons(tail, nil)) \wedge \neg\{head\} \subseteq \{tail\} \wedge ispath(cons(tail, nil))$
$p_{tail_i} = getp_{\mathsf{K}}(h, head, tail, i)$	$h[head].next[i] = tail \wedge isreachp_{\mathsf{K}}(h, tail, tail, i, nil)$
$s_{tail_i} = path2set(p_{tail_i})$	$s_{tail_i} = \{head\}$

Once we have achieved this point, we only need to call the decision procedures for each of the involved theories in T_{SLKBase}. ∗

5.3 Extending TSL$_\mathsf{K}$ to Reason about (L, U, H)

In Section 3.2 we presented the algorithms for INSERT, REMOVE and SEARCH over a concurrent skiplist. In that moment, we also annotated the code with ghost arrays L, U and H. L and U are arrays from thread identifiers to addresses while H is an array from thread identifiers to skiplist levels. The idea is that the triplet $(L[t], U[t], H[t])$ denotes the bounds of the section of the skiplist that thread t can modify. This is not a restriction imposed to thread t, but just a consequence of the progress of t. We usually refer to such section of the skiplist as the (L, U, H) of thread t.

Following these algorithms, in this section we show how we can extend TSL$_\mathsf{K}$ with some extra functions to let us reason about (L, U, H) inclusion. When we presented the pessimistic version of programs INSERT, REMOVE and SEARCH we stated that such implementation enjoys fairness. In fact, we can verify that any thread whose (L, U, H) does not strictly contain the (L, U, H) of another thread is guaranteed to terminate.

To begin with, we present two new theories we will use. A theory of pairs of thread identifiers (T_{pair}) and a theory of sets of pairs (T_{setpair}). The signature of these two new theories is defined as one may expect:

$$
\begin{array}{lcl}
\Sigma_{\mathsf{pair}} & = & \big(\{\ \mathsf{pair}, \mathsf{thid} \qquad\qquad\qquad\qquad\qquad\qquad\ \},\\
& & \ \{\ \langle \_, \_ \rangle \ :\ \mathsf{thid} \times \mathsf{thid} \to \mathsf{pair} \qquad\ \},\\
& & \ \{\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \}\ \big)
\end{array}
$$

$$
\begin{array}{lcl}
\Sigma_{\mathsf{setpair}} & = & \big(\{\ \mathsf{setpair}, \mathsf{pair} \qquad\qquad\qquad\qquad\qquad\ \},\\
& & \ \{\ \emptyset_{\mathsf{P}} \ :\ \mathsf{setpair},\\
& & \quad \{\_\}_{\mathsf{P}} \ :\ \mathsf{pair} \to \mathsf{setpair},\\
& & \quad \cup_{\mathsf{P}} \ :\ \mathsf{setpair} \times \mathsf{setpair} \to \mathsf{setpair},\\
& & \quad \cap_{\mathsf{P}} \ :\ \mathsf{setpair} \times \mathsf{setpair} \to \mathsf{setpair},\\
& & \quad \setminus_{\mathsf{P}} \ :\ \mathsf{setpair} \times \mathsf{setpair} \to \mathsf{setpair}\ \},\\
& & \ \{\ \in_{\mathsf{P}} \ :\ \mathsf{pair} \times \mathsf{setpair}\\
& & \quad \subseteq_{\mathsf{P}} \ :\ \mathsf{setpair} \times \mathsf{setpair} \qquad\ \}\ \big)
\end{array}
$$

The interpretation for each new function and predicate is the expected one for pairs and sets. $\mathcal{A}_{\mathsf{pair}} = \mathcal{A}_{\mathsf{thid}} \times \mathcal{A}_{\mathsf{thid}}$ while $\mathcal{A}_{\mathsf{setpair}}$ is the power-set of $\mathcal{A}_{\mathsf{pair}}$. Besides, we introduce a new theory, T_{LUH}, to reason about (L, U, H). The signature of T_{LUH}, Σ_{LUH}, is given by:

$$
\begin{array}{lcl}
\Sigma_{\mathsf{LUH}} & = & \big(\{\ \mathsf{mem}, \mathsf{aarr}, \mathsf{larr}, \mathsf{thid}, \mathsf{settid}, \mathsf{pair}, \mathsf{setpair} \qquad\qquad\qquad\qquad\ \},\\
& & \ \{\ \mathit{subset}_{LUH} \ :\ \mathsf{mem} \times \mathsf{aarr} \times \mathsf{aarr} \times \mathsf{larr} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_\mathsf{K} \to \mathsf{settid}\\
& & \quad \mathit{inter}_{LUH} \ :\ \mathsf{mem} \times \mathsf{aarr} \times \mathsf{aarr} \times \mathsf{larr} \times \mathsf{settid} \to \mathsf{setpair}\\
& & \quad \mathit{empty}_{LUH} \ :\ \mathsf{mem} \times \mathsf{aarr} \times \mathsf{aarr} \times \mathsf{larr} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_\mathsf{K} \to \mathsf{settid}\\
& & \quad \mathit{min}_{LUH} \ :\ \mathsf{mem} \times \mathsf{aarr} \times \mathsf{settid} \to \mathsf{thid} \qquad\qquad\qquad\qquad\qquad\ \},\\
& & \ \{\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \}\ \big)
\end{array}
$$

Function subset_{LUH} receives as argument a memory, three arrays (representing the (L, U, H) regions for a skiplist), a couple of addresses and a level. It returns the set of thread identifiers whose (L, U, H) is strictly contained into the region bounded by the address and level taken as argument. Function inter_{LUH}

receives as argument a set of pairs and it returns the subset of pairs whose (L, U, H) intersects with each other. Function $empty_{LUH}$ is similar to $subset_{LUH}$, except that it returns the set of thread identifiers whose (L, U, H) is empty, that is, those whose (L, U, H) does not contain the (L, U, H) of any other thread. Finally, function min_{LUH} receives a set of thread id and returns the minimum of all of them. We say that the minimum is the one with its right bound closest to the tail of the skiplist. To simplify notation, if a_1 and a_2 are two variables of type address, considering a memory m, we use

$$a_1 \leq a_2 \qquad \text{for} \qquad a_2 \in addr2set_{\mathsf{K}}(m, a_1, 0)$$

Similarly, if $\mathcal{A}$ is a model, i_1 and i_2 belong to $\mathcal{A}_{\mathsf{addr}}$ and $m \in \mathcal{A}_{\mathsf{mem}}$, we abuse notation and write

$$i_1 \leq i_2 \qquad \text{for} \qquad i_2 \in addr2set_{\mathsf{K}}{}^{\mathcal{A}}(m, i_1, 0)$$

Using the syntactic sugar we have just introduced, given a model $\mathcal{A}$, the functions of Σ_{LUH} are interpreted this way:

$$\begin{aligned}
subset_{LUH}{}^{\mathcal{A}}(m, L, U, H, a_1, a_2, l) &= \{\ t' \in \mathcal{A}_{\mathsf{thid}} \mid a_1 \leq L(t') \text{ and } U(t') \leq a_2 \text{ and } \\
&\qquad (H(t') < l \text{ or } H(t') = l)\ \} \\
inter_{LUH}{}^{\mathcal{A}}(m, L, U, H, g) &= \{\ \langle t_1, t_2\rangle \in \mathcal{A}_{\mathsf{pair}} \mid t_1 \in g, t_2 \in g, \\
&\qquad intersects(t_1, t_2) \text{ and } t_1 \neq t_2\ \} \\
empty_{LUH}{}^{\mathcal{A}}(m, L, U, H, a_1, a_2, l) &= \{\ t' \in \mathcal{A}_{\mathsf{thid}} \mid t' \in subset_{LUH}{}^{\mathcal{A}}(m, L, U, H, a_1, a_2, l) \text{ and } \\
&\qquad subset_{LUH}{}^{\mathcal{A}}(m, L, U, H, L(t'), U(t'), H(t')) = \emptyset\ \} \\
min_{LUH}{}^{\mathcal{A}}(m, U, g) &= \begin{cases} \oslash & \text{if } g = \emptyset \\ t_i & \text{if } g = \{t_1, \ldots, t_q\},\ i \in 1..q \text{ and for all } j \in 1..q\ U(t_i) \leq U(t_j) \end{cases}
\end{aligned}$$

for each $m \in \mathcal{A}_{\mathsf{mem}}$; $L, U \in \mathcal{A}_{\mathsf{aarr}}$; $H \in \mathsf{larr}$; $a_1, a_2 \in \mathcal{A}_{\mathsf{addr}}$; $l \in \mathcal{A}_{\mathsf{level_K}}$; $w \in \mathcal{A}_{\mathsf{setpair}}$ and $g \in \mathcal{A}_{\mathsf{settid}}$. Besides, we define *intersects* by:

$$\begin{aligned}
intersects(t_1, t_2) \;=\; & (\ L(t_2) \leq L(t_1) \ \wedge\ L(t_1) \leq U(t_2) \ \wedge\ H(t_1) \geq 1 \wedge H(t_2) \geq 1\ ) \ \vee \\
& (\ L(t_2) \leq U(t_1) \ \wedge\ U(t_1) \leq U(t_2) \ \wedge\ H(t_1) \geq 1 \wedge H(t_2) \geq 1\ ) \ \vee \\
& (\ L(t_1) \leq L(t_2) \ \wedge\ U(t_2) \leq U(t_1) \ \wedge\ H(t_2) > H(t_1)\ )
\end{aligned}$$

We now define the theory TSL_K+ as the one obtained from the union of TSL_K with T_{pair}, T_{setpair} and T_{LUH}. As before, we now define the set of TSL_K+-normalized literals.

Definition 5.6 (TSL_K+-normalized literals).
A TSL_K+-literal is normalized if it is a TSL_K-normalized literal or it is a flat literal of the form:

$$\begin{array}{rclcrcl}
u &=& \langle t_1, t_2\rangle & & & & \\
w &=& \{u\}_{\mathrm{P}} & & w_1 &=& w_2 \cup_{\mathrm{P}} w_3 \\
w_1 &=& w_2 \setminus_{\mathrm{P}} w_3 & & & & \\
g &=& subset_{LUH}(m, L, U, H, a_1, a_2, l) & & w &=& inter_{LUH}(m, L, U, H, g) \\
g &=& empty_{LUH}(m, L, U, H, a_1, a_2, l) & & t &=& min_{LUH}(m, U, g)
\end{array}$$

where a_1 and a_2 are addr*-variables; m is a* mem*-variable; g is a* settid*-variable; L and U are* aarr*-variables; H is a Σ_{larr}-variable; l is a $\mathsf{level_K}$-variable; t_1 and t_2 are* thid*-variables; u is* pair*-variable and w, w_1, w_2 and w_3 are* setpair*-variables.* †

The non normalized literals can be constructed from the normalized ones using an approach similar to the one employed in the definition of literals from sets of thread identifiers. We now show that TSL_K+ still satisfies the finite model property.

Lemma 5.5 (Finite Model Property for TSL$_K$+):
Let Γ be a conjunction of normalized TSL$_K$+-literals. Let $\overline{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\overline{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\overline{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\overline{p} = |V_{\mathsf{path}}(\Gamma)|$, $\overline{t} = |V_{\mathsf{thid}}(\Gamma)|$, $\overline{o} = |V_{\mathsf{ord}}(\Gamma)|$, $\overline{u} = |V_{\mathsf{pair}}(\Gamma)|$ and $\overline{x} = |V_{\mathsf{aarr}}(\Gamma)|$. Then the following are equivalent:

1. *Γ is TSL$_K$+-satisfiable;*
2. *Γ is true in a TSL$_K$+ interpretation $\mathcal{B}$ such that*

$$\begin{array}{rcl} |\mathcal{B}_{\mathsf{level_K}}| & \leq & K \\ |\mathcal{B}_{\mathsf{thid}}| & \leq & \overline{t} + 1 + K\,\overline{m}\,\overline{a} + 2\,\overline{u} \\ |\mathcal{B}_{\mathsf{addr}}| & \leq & \overline{a} + 1 + K\,\overline{m}\,\overline{a} + \overline{p}^2 + \overline{p}^3 + 2\,\overline{m}\,\overline{p} + \overline{x}\,|\mathcal{B}_{\mathsf{thid}}| \\ |\mathcal{B}_{\mathsf{elem}}| & \leq & \overline{e} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}| \\ |\mathcal{B}_{\mathsf{ord}}| & \leq & \overline{o} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}| \end{array}$$

♠

Proof $(2 \rightarrow 1)$ is immediate.

$(1 \rightarrow 2)$.

We prove this implication only for the new TSL$_K$+-literals. Let $\mathcal{A}$ be a TSL$_K$-interpretation satisfying a set of normalized TSL$_K$+-literals Γ. We use $\mathcal{A}$ to construct a finite TSL$_K$+-interpretation $\mathcal{B}$. $\mathcal{B}$ is construted as for TSL$_K$, except for the set of thread identifiers, which is defined as

$$\begin{array}{rcll} \mathcal{B}_{\mathsf{thid}} & = & V_{\mathsf{thid}}^{\mathcal{A}} \cup \left\{ \oslash \right\} & \cup \\ & & \left\{ m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}(l) \mid m \in V_{\mathsf{mem}}, a \in V_{\mathsf{addr}} \text{ and } l \in \mathcal{B}_{\mathsf{level_K}} \right\} & \cup \\ & & \left\{ t_1, t_2 \in \mathcal{A}_{\mathsf{thid}} \mid (t_1,t_2) = p^{\mathcal{A}} \text{ and } p \in V_{\mathsf{pair}} \right\} & \end{array}$$

and the interpretation of the new variables is given by

$$\begin{array}{rcll} u^{\mathcal{B}} & = & u^{\mathcal{A}} & \text{for each } u \in V_{\mathsf{pair}} \\ w^{\mathcal{B}} & = & w^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) & \text{for each } w \in V_{\mathsf{setpair}} \end{array}$$

We now proceed to the analysis of the new literals:

Literals of the form $u = \langle t_1, t_2 \rangle$:

$$\begin{array}{rcl} u^{\mathcal{B}} & = & u^{\mathcal{A}} \\ & = & ({t_1}^{\mathcal{A}}, {t_2}^{\mathcal{A}}) \\ & = & ({t_1}^{\mathcal{B}}, {t_2}^{\mathcal{B}}) \end{array}$$

Literals of the form $w = \{u\}_{\mathrm{P}}$:

$$\begin{array}{rcl} w^{\mathcal{B}} & = & w^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\ & = & \{u^{\mathcal{A}}\}_{\mathrm{P}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\ & = & \{u^{\mathcal{B}}\}_{\mathrm{P}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\ & = & \{u^{\mathcal{B}}\}_{\mathrm{P}} \end{array}$$

Literals of the form $w_1 = w_2 \cup_{\mathrm{P}} w_3$:

$$\begin{array}{rcl} w_1^{\mathcal{B}} & = & w_1^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\ & = & \left(w_2^{\mathcal{A}} \cup w_3^{\mathcal{A}} \right) \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\ & = & \left(w_2^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \right) \cup \left(w_3^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \right) \\ & = & w_2^{\mathcal{B}} \cup_{\mathrm{P}} w_3^{\mathcal{B}} \end{array}$$

Literals of the form $w_1 = w_2 \setminus_{\mathrm{P}} w_3$:

$$\begin{aligned}
w_1^{\mathcal{B}} &= w_1^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\
&= \left(w_2^{\mathcal{A}} \setminus w_3^{\mathcal{A}}\right) \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\
&= \left(w_2^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}})\right) \setminus \left(w_3^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}})\right) \\
&= w_2^{\mathcal{B}} \setminus_{\mathrm{P}} w_3^{\mathcal{B}}
\end{aligned}$$

Literals of the form $g = subset_{LUH}(m, L, U, H, a_1, a_2, l)$:

$$\begin{aligned}
g^{\mathcal{B}} &= g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} \\
&= subset_{LUH}{}^{\mathcal{A}}(m^{\mathcal{A}}, L^{\mathcal{A}}, U^{\mathcal{A}}, H^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}}) \cap \mathcal{B}_{\mathsf{thid}} \\
&= \left\{t' \in \mathcal{A}_{\mathsf{thid}} \mid a_1^{\mathcal{A}} \leq L^{\mathcal{A}}(t') \text{ and } U^{\mathcal{A}}(t') \leq a_2^{\mathcal{A}} \text{ and } (H^{\mathcal{A}}(t') < l^{\mathcal{A}} \text{ or } H^{\mathcal{A}}(t') = l^{\mathcal{A}})\right\} \cap \mathcal{B}_{\mathsf{thid}} \\
&= \left\{t' \in \mathcal{B}_{\mathsf{thid}} \mid a_1^{\mathcal{B}} \leq L^{\mathcal{B}}(t') \text{ and } U^{\mathcal{B}}(t') \leq a_2^{\mathcal{B}} \text{ and } (H^{\mathcal{B}}(t') < l^{\mathcal{B}} \text{ or } H^{\mathcal{B}}(t') = l^{\mathcal{B}})\right\} \\
&= subset_{LUH}{}^{\mathcal{B}}(m^{\mathcal{A}}, L^{\mathcal{B}}, U^{\mathcal{B}}, H^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}})
\end{aligned}$$

Literals of the form $w = inter_{LUH}(m, L, U, H, g)$:

$$\begin{aligned}
w^{\mathcal{B}} &= w^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\
&= inter_{LUH}{}^{\mathcal{A}}(m^{\mathcal{A}}, L^{\mathcal{A}}, U^{\mathcal{A}}, H^{\mathcal{A}}, g_1^{\mathcal{A}}) \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\
&= \left\{\langle t_1, t_2\rangle \in \mathcal{A}_{\mathsf{pair}} \mid t_1 \in g^{\mathcal{A}}, t_2 \in g^{\mathcal{A}}, intersects(t_1, t_2) \text{ and } t_1 \neq t_2\right\} \cap (\mathcal{B}_{\mathsf{thid}} \times \mathcal{B}_{\mathsf{thid}}) \\
&= \left\{\langle t_1, t_2\rangle \in \mathcal{B}_{\mathsf{pair}} \mid t_1 \in g^{\mathcal{B}}, t_2 \in g^{\mathcal{B}}, intersects(t_1, t_2) \text{ and } t_1 \neq t_2\right\} \\
&= inter_{LUH}{}^{\mathcal{B}}(m^{\mathcal{A}}, L^{\mathcal{B}}, U^{\mathcal{B}}, H^{\mathcal{B}}, g^{\mathcal{B}})
\end{aligned}$$

Literals of the form $g = empty_{LUH}(m, L, U, H, a_1, a_2, l)$:

$$\begin{aligned}
g^{\mathcal{B}} &= g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} \\
&= empty_{LUH}{}^{\mathcal{A}}(m^{\mathcal{A}}, L^{\mathcal{A}}, U^{\mathcal{A}}, H^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}}) \cap \mathcal{B}_{\mathsf{thid}} \\
&= \Big\{t' \in \mathcal{A}_{\mathsf{thid}} \mid t' \in subset_{LUH}{}^{\mathcal{A}}(m^{\mathcal{A}}, L^{\mathcal{A}}, U^{\mathcal{A}}, H^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}}) \text{ and} \\
&\qquad\qquad subset_{LUH}{}^{\mathcal{A}}(m^{\mathcal{A}}, L^{\mathcal{A}}, U^{\mathcal{A}}, H^{\mathcal{A}}, L^{\mathcal{A}}(t'), U^{\mathcal{A}}(t'), H^{\mathcal{A}}(t')) = \emptyset\Big\} \cap \mathcal{B}_{\mathsf{thid}} \\
&= \Big\{t' \in \mathcal{B}_{\mathsf{thid}} \mid t' \in subset_{LUH}{}^{\mathcal{B}}(m^{\mathcal{B}}, L^{\mathcal{B}}, U^{\mathcal{B}}, H^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}}) \text{ and} \\
&\qquad\qquad subset_{LUH}{}^{\mathcal{B}}(m^{\mathcal{A}}, L^{\mathcal{B}}, U^{\mathcal{B}}, H^{\mathcal{B}}, L^{\mathcal{B}}(t'), U^{\mathcal{B}}(t'), H^{\mathcal{B}}(t')) = \emptyset\Big\} \\
&= empty_{LUH}{}^{\mathcal{B}}(m^{\mathcal{B}}, L^{\mathcal{B}}, U^{\mathcal{B}}, H^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}})
\end{aligned}$$

Literals of the form $t = min_{LUH}(m, U, g)$:

$$\begin{aligned}
t^{\mathcal{B}} &= t^{\mathcal{A}} \\
&= min_{LUH}{}^{\mathcal{A}}\ (m^{\mathcal{A}},\ U^{\mathcal{A}},\ g^{\mathcal{A}})
\end{aligned}$$

Then, we have two cases to consider. If $g^{\mathcal{A}} = \emptyset$, then $g^{\mathcal{B}} = g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} = \emptyset \cap \mathcal{B}_{\mathsf{thid}} = \emptyset$. Besides, $\oslash = t^{\mathcal{A}} = t^{\mathcal{B}} = min_{LUH}{}^{\mathcal{B}}(m^{\mathcal{B}}, U^{\mathcal{B}}, g^{\mathcal{B}})$. So, let's now consider the other possible case. Let's assume $g = \{t_1, \ldots, t_q\}$. Let $t_i \in g$ be a thread such that for all $j \in 1..q$, $U(t_j) \leq U(t_i)$. At $t \in V_{\mathsf{thid}}$, $t^{\mathcal{A}} = t^{\mathcal{B}} = t_i^{\mathcal{B}} \in \mathcal{B}_{\mathsf{thid}}$. Besides, $g^{\mathcal{B}} = g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}}$ and hence $t_i^{\mathcal{B}} \in g^{\mathcal{B}}$. Therefore, $t_i^{\mathcal{B}} = min_{LUH}(m^{\mathcal{B}}, U^{\mathcal{B}}, g^{\mathcal{B}})$. □

5.4 Verifying Some Properties Over Concurrent Skiplists

In this section we show how some of the properties we would like to describe about a skiplist are translated to TSL$_K$, so that the decision procedure described in this chapter can be used to solve them.

5.4.1 Skiplist Preservation

In Example 3.4 we gave an idea of the functions and predicates we would require to verify skiplists presenting the verification condition for transition 43 of program Insert. Such verification condition applied to a skiplist of height 4 was $SkipList_4(h, sl.head, sl.tail) \wedge \phi \rightarrow SkipList_4(h', sl'.head, sl'.tail)$, where ϕ is:

$$\begin{pmatrix} x.key = newval \wedge \\ prev.key < newval \wedge \\ x.next[i].key > newval \wedge \\ prev.next[i] = x.next[i] \wedge \\ (x,i) \notin sl.r \wedge 0 \leq i \leq 3 \end{pmatrix} \wedge \begin{pmatrix} at_{43}^{[j]} \wedge \\ prev'.next[i] = x \wedge \\ at'^{[j]}_{44} \wedge \\ h' = h \wedge sl = sl' \wedge \\ x' = x \quad \dots \end{pmatrix}$$

We now show how to write this verification condition using TSL$_K$ so that the decision procedure can be applied. We assume x : addr and $prev$: addr. Besides, for the sake of simplicity, we assume variables $head$: addr, $tail$: addr and r : mrgn such that $head = sl.head$, $tail = sl.tail$ and $r = sl.r$.

Remember we are working with parametrized systems. Hence, when we use x, $prev$ or $newval$, in fact we mean the local copy of these variables for a particular thread, let's say, thread j. Then we are in fact working with $x^{[j]}$, $prev^{[j]}$ and $newval^{[j]}$ respectively. We are also assuming that proofs are done for a closed system, after the number of threads involved in the system has been fixed. This way, we assume reasoning over a finite and bounded number of threads. In more general parametrized systems, one can reason over an unbounded number of threads. In such case, a theory of arrays equipped with a decision

Original term	TSL$_K$ term
$ordList(h, head, 0)$	$\overbrace{p = getp_K(h, head, null, 0)}^{\Sigma_{bridge}} \wedge \overbrace{ordList(h, p)}^{\Sigma_{bridge}}$
$tail.next[0] = null$ $\wedge \cdots \wedge$ $tail.next[3] = null$	$\overbrace{c_{tail} = h[tail]}^{\Sigma_{mem}} \wedge \overbrace{a_{tail_0} = c_{tail}.next[0]}^{\Sigma_{cell}} \wedge \overbrace{a_{tail_0} = null}^{\Sigma_{addr}}$ $\wedge \quad \vdots \quad \wedge \quad \vdots$ $\overbrace{a_{tail_3} = c_{tail}.next[3]}^{\Sigma_{cell}} \wedge \overbrace{a_{tail_3} = null}^{\Sigma_{addr}}$
$SubList(h, head, tail, 1,$ $head, tail, 0)$ $\wedge \cdots \wedge$ $SubList(h, head, tail, 3,$ $head, tail, 2)$	$\overbrace{p_{tail_0} = getp_K(h, head, tail, 0)}^{\Sigma_{bridge}} \wedge \cdots \wedge \overbrace{p_{tail_3} = getp_K(h, head, tail, 3)}^{\Sigma_{bridge}} \wedge$ $\overbrace{s_{tail_0} = path2set(p_{tail_0})}^{\Sigma_{bridge}} \wedge \cdots \wedge \overbrace{s_{tail_3} = path2set(p_{tail_3})}^{\Sigma_{bridge}} \wedge$ $\overbrace{s_{tail_1} \subseteq s_{tail_0}}^{\Sigma_{set}} \wedge \cdots \wedge \overbrace{s_{tail_3} \subseteq s_{tail_2}}^{\Sigma_{set}}$

Figure 5.6: Translation of $SkipList_4(h, sl.head, sl.tail)$ into TSL$_K$

Original term	TSL_K term
$x.key = newval$	$\overbrace{c_x = h[x]}^{\Sigma_{\text{mem}}} \wedge \overbrace{c_x.key = newval}^{\Sigma_{\text{cell}}}$
$prev.key < newval$	$\overbrace{c_{prev} = h[prev]}^{\Sigma_{\text{mem}}} \wedge \overbrace{c_{prev}.key = k_{prev}}^{\Sigma_{\text{cell}}} \wedge \overbrace{k_{prev} \preceq newval}^{\Sigma_{\text{ord}}} \wedge \overbrace{k_{prev} \neq newval}^{\Sigma_{\text{ord}}}$
$prev.next[i].key > newval$	$\overbrace{a_{next} = c_{prev}.next[i]}^{\Sigma_{\text{cell}}} \wedge \overbrace{c_{next} = h[a_{next}]}^{\Sigma_{\text{mem}}} \wedge \overbrace{k_{next} = c_{next}.key}^{\Sigma_{\text{cell}}} \wedge \overbrace{newval \preceq k_{next}}^{\Sigma_{\text{ord}}} \wedge \overbrace{newval \neq k_{next}}^{\Sigma_{\text{ord}}}$
$prev.next[i] = x.next[i]$	$\overbrace{a_{next_x} = c_x.next[i]}^{\Sigma_{\text{cell}}} \wedge \overbrace{a_{next} = a_{next_x}}^{\Sigma_{\text{addr}}}$
$prev.lockid[i] = j$	$\overbrace{t_{prev} = c_{prev}.lockid[i]}^{\Sigma_{\text{cell}}} \wedge \overbrace{t_{prev} = j}^{\Sigma_{\text{thid}}}$
$x.next[i].lockid[i] = j$	$\overbrace{c_{next_x} = h[a_{next_x}]}^{\Sigma_{\text{mem}}} \wedge \overbrace{t_{next_x} = c_{next_x}.lockid[i]}^{\Sigma_{\text{cell}}} \wedge \overbrace{t_{next_x} = j}^{\Sigma_{\text{thid}}}$
$(x, i) \notin r$	$\overbrace{empty = \mathbf{emp}_{\text{MR}}}^{\Sigma_{\text{mrgn}}} \wedge \overbrace{x_{i_{set}} = \langle x, i \rangle_{\text{MR}}}^{\Sigma_{\text{mrgn}}} \wedge \overbrace{empty = r \cap_{\text{MR}} x_{i_{set}}}^{\Sigma_{\text{mrgn}}}$
$0 \leq i \leq 3$	We must generate four different scenarios to be combined with the conjunction of the remaining literals. These scenarios are: 1. $\overbrace{0 < i}^{\Sigma_{\text{level}_\text{K}}} \wedge \overbrace{i < 3}^{\Sigma_{\text{level}_\text{K}}}$ 2. $\overbrace{0 < i}^{\Sigma_{\text{level}_\text{K}}} \wedge \overbrace{i = 3}^{\Sigma_{\text{level}_\text{K}}}$ 3. $\overbrace{0 = i}^{\Sigma_{\text{level}_\text{K}}} \wedge \overbrace{i < 3}^{\Sigma_{\text{level}_\text{K}}}$ 4. $\overbrace{0 = i}^{\Sigma_{\text{level}_\text{K}}} \wedge \overbrace{i = 3}^{\Sigma_{\text{level}_\text{K}}}$

Figure 5.7: Translation of enabling conditions into TSL_K

Original term	TSL$_K$ term
$at_{43}^{[j]} \wedge at'^{[j]}_{44}$	$\overbrace{at_{j_{43}}}^{\text{boolean}} \wedge \overbrace{\neg at_{j_{44}}}^{\text{boolean}} \wedge \overbrace{\neg at'_{j_{43}}}^{\text{boolean}} \wedge \overbrace{at'_{j_{44}}}^{\text{boolean}}$
$prev'.next[i] = x$	$\overbrace{a_{next_{new}} = c_{prev_{new}}.next[i]}^{\Sigma_{\text{cell}}} \wedge \overbrace{a_{next_{new}} = x}^{\Sigma_{\text{addr}}}$
$prev'.key = prev.key$	$\overbrace{k_{prev_{new}} = c_{prev_{new}}.key}^{\Sigma_{\text{cell}}} \wedge \overbrace{k_{prev} = k_{prev_{new}}}^{\Sigma_{\text{ord}}}$
$prev'.data = prev.data$	$\overbrace{e_{prev} = c_{prev}.data}^{\Sigma_{\text{cell}}} \wedge \overbrace{e_{prev_{new}} = c_{prev_{new}}.data}^{\Sigma_{\text{cell}}} \wedge \overbrace{e_{prev} = e_{prev_{new}}}^{\Sigma_{\text{elem}}}$
$prev'.next[0] = prev.next[0] \wedge \cdots \wedge prev'.next[i-1] = prev.next[i-1] \wedge prev'.next[i+1] = prev.next[i+1] \wedge \cdots \wedge prev'.next[3] = prev.next[3]$	$\overbrace{a_{prev_0} = c_{prev}.next[0]}^{\Sigma_{\text{cell}}} \wedge \cdots \wedge \overbrace{a_{prev_{i-1}} = c_{prev}.next[i-1]}^{\Sigma_{\text{cell}}} \wedge \overbrace{a_{prev_{i+1}} = c_{prev}.next[i+1]}^{\Sigma_{\text{cell}}} \wedge \cdots \wedge \overbrace{a_{prev_3} = c_{prev}.next[3]}^{\Sigma_{\text{cell}}} \wedge \overbrace{a_{prev_{new_0}} = c_{prev_{new}}.next[0]}^{\Sigma_{\text{cell}}} \wedge \cdots \wedge \overbrace{a_{prev_{new_{i-1}}} = c_{prev_{new}}.next[i-1]}^{\Sigma_{\text{cell}}} \wedge \overbrace{a_{prev_{new_{i+1}}} = c_{prev_{new}}.next[i+1]}^{\Sigma_{\text{cell}}} \wedge \cdots \wedge \overbrace{a_{prev_{new_3}} = c_{prev_{new}}.next[3]}^{\Sigma_{\text{cell}}} \wedge \overbrace{a_{prev_{new_0}} = a_{prev_0}}^{\Sigma_{\text{addr}}} \wedge \cdots \wedge \overbrace{a_{prev_{new_{i-1}}} = a_{prev_{i-1}}}^{\Sigma_{\text{addr}}} \wedge \overbrace{a_{prev_{new_{i+1}}} = a_{prev_{i+1}}}^{\Sigma_{\text{addr}}} \wedge \cdots \wedge \overbrace{a_{prev_{new_3}} = a_{prev_3}}^{\Sigma_{\text{addr}}} \wedge \overbrace{h' = upd(h, prev, c_{prev_{new}})}^{\Sigma_{\text{mem}}}$

Figure 5.8: Translation of the modifications introduced by the transition relation into TSL$_K$

procedure [BMS06] can be used to keep the values of all local variables and program counters as arrays. Moreover, such theory is easily combined with TSL$_K$. However, this approach goes beyond the scope of this work.

Fig. 5.6 contains the translation of the $SkipList_4$ predicate into literals of TSL$_K$. The translation for predicate $SkipList_4(h', sl'.head, sl'.tail)$ is very similar to $SkipList_4(h, sl.head, sl.tail)$ and thus it is omitted. The difference between them resides only on the modifications introduced by the transition relation, stating $SkipList_4(h', sl'.head, sl'.tail)$ that the structure keeps the shape of a skiplist after the transition has been taken. Fig. 5.7 describes the translation of the enabling conditions for the presented verification condition into TSL$_K$. Similarly, Fig. 5.8 depicts the translation of the modifications produced by the transition. In all cases, it is straightforward to realize that the resulting literals are all in TSL$_K$ and thus it is possible to check their satisfiability using the decision procedure described in this chapter. For the moment, this translation must be accomplished manually. Future work includes the construction of a

tool to automatically translate formulas into TSL_K.

When describing the new literals in TSL_K obtained from the verification condition, some fresh variables must be introduced. To describe them we use a for variables of sort addr, k for variables of sort ord, e for variables of sort elem, t for variables of sort thid, c for variables of sort cell, p for variables of sort path and s for variables of sort set.

5.4.2 Termination of an Arbitrary Thread

When we introduced concurrent skiplists in Chapter 3, we showed that some implementations do not guarantee termination of all threads if we do not assume a fair scheduler. Because of this, we introduced a new implementation we called *pessimistic*. Now, we sketch how the termination of an arbitrary thread running the pessimistic implementation of a concurrent skiplist (presented in Section 3.2) could be proved.

When we gave the implementations, we enriched the algorithms with ghost arrays L, U and H, to define the (L, U, H) section for each running thread. Remember that (L, U, H) represents the maximum portion of the skiplist we are sure a thread can potentially modify.

The pessimistic version left a locked node behind, to prevent threads to overtake other threads. First of all, we extend the definition of skiplist to capture the property that, in pessimistic skiplists, two (L, U, H) belonging to different threads are disjoint or one is strictly contained into the other. For such purpose, we define

$$\begin{aligned} Skiplist^{pessi}_{\mathsf{K}}(h, sl : SkipList) \quad \hat{=} \quad & SkipList_{\mathsf{K}}(h, sl.head, sl.tail) \ \wedge \\ & \emptyset_{\mathsf{P}} = inter_{LUH}(m, L, U, H, subset_{LUH}(m, L, U, H, sl.head, sl.tail, \mathsf{K}-1)) \end{aligned}$$

The new predicate $Skiplist^{pessi}_{\mathsf{K}}(h, sl)$ says that in heap h, sl points to a structure with the shape of a skiplist and no (L, U, H) within the skiplist partially overlays with other thread's (L, U, H).

Imagine we want to prove that an arbitrary thread k terminates. Here we do not give the diagram, but just the intuition behind the proof. Let $(L, U, H)_k$ be the (L, U, H) of thread k. A key property is that the number of threads whose (L, U, H) is contained into $(L, U, H)_k$ does not increment. In fact, we can use the cardinality of $subset_{LUH}(m, L, U, H, L(k), U(k), H(k))$ as a ranking function to ensure that no new thread is entering $(L, U, H)_k$. Besides, we can ensure that it always exists a thread that has no contention when trying to progress. Such thread is the one with the minimum (L, U, H) within $(L, U, H)_k$, that is, thread t_{min} where

$$t_{min} = min_{LUH}(m, U, empty_{LUH}(m, L, U, H, L(k), U(k), H(k), \mathsf{K}-1)).$$

6

Conclusion

In this work, we have presented combinable decision procedures for concurrent lists and skiplists. The development of such decision procedures is motivated by the necessity of automating the verification process of temporal properties (safety and liveness) of an imperative implementation of concurrent datatypes. Although we have focused on concurrent lists and skiplists, the approach can be applied to the verification of more complex concurrent data structures.

As shown, the verification is performed using verification diagrams – a complete method to prove temporal properties of reactive systems – and explicit reasoning on memory regions. The verification process usually requires the aid of ghost variables. Verification is reduced to proving a finite number of verification conditions, which requires decision procedures in the appropriate theories, including regions, pointers, locks and specific theories for memory layouts (in this case single linked-lists and skiplists).

There are some key differences between our approach and other approaches in the literature. Building on the success of separation logic in proving sequential programs, the most popular approach has consisted in extending separation logic to concurrent programs. These extensions require adapting techniques like rely-guarantee that cannot be directly used with separation logic. Our decision to use explicit regions (finite sets of addresses) allows the direct use of classical techniques like assume-guarantee and the combination of decision procedures. Furthermore, in concurrent separation logic, it is critical to describe memory footprints of sections of code. This description becomes very cumbersome when the code is not organized in mutual exclusion regions, as in fine-grain synchronization algorithms. Moreover, the integration into SMT solvers is quite straightforward with classical logics, but it is still an open question with separation logic.

The technique we propose can be seen as a method to separate the reasoning about concurrency (with verification diagrams) from the reasoning about the memory (with decision procedures). The former is independent of the data structure under consideration. We are currently extending our approach to the verification of other pointer-based concurrent data structures like concurrent hash maps. Again, sharing in these data structures makes it very hard to reason using separation logic. For our approach, these extensions will require the design of suitable decision procedures.

We have also presented TLL3 and TSL_K, theories of concurrent singly linked lists and concurrent skiplists of height at most K respectively. We showed both of them are useful for automatically proving the VCs generated during the verification of concurrent list and skiplist implementations.

TLL3 is capable of reasoning about memory, cells, pointers and reachability. TSL_K can also reason about masked regions, enabling ordered lists and sublists, allowing the description of the skiplist property as well as the representation of memory modifications introduced by the execution of program statements.

We showed both theories decidable by proving their finite model property and exhibiting the minimal cardinality of a model if one such model exists. Moreover, we showed how to reduce the satisfiability problem of quantifier-free TLL3 and TSL_K formulas to a combination of theories using the many-sorted

version of Nelson-Oppen, allowing the use of well studied decision procedures. The complexity of the decision problem for TLL3 and TSL_K is easily shown to be NP-complete since they properly extend TLL [RZ06a].

Current work includes among other things the translation of formulas from T_{ord}, $T_{\mathsf{level}_\mathsf{k}}$, T_{set}, T_{settid} and T_{mrgn} into BAPA [KNR05]. In BAPA, arithmetic, sets and cardinality aid in the definition of skiplists properties. At the same time, paths can be represented as finite sequences of addresses. We are studying how to replace the recursive functions from T_{reach} and Σ_{bridge} by canonical set and list abstractions [SDK10], which would lead to a more efficient decision procedure, essentially encoding full TLL3 and TSL_K formulas into BAPA.

In the case of TSL_K, the family of theories presented in the work is limited to skiplists of a fixed maximum height. Typical skiplist implementations fix a maximum number of levels and this can be handled with TSL_K. Inserting more than 2^{levels} elements into a skiplist may slow-down the search of a skiplist implementation but this issue affects performance and not correctness, which is the goal pursued in this paper. Following this line, we are studying techniques to describe skiplists of arbitrary many levels. A promising approach consists of reducing a formula describing an unbounded number of levels to a formula grounded on a finite number of them, reducing the satisfiability problem to TSL_K. This approach, however, is still work in progress.

Future work also includes building a generic verification condition generator for verification diagrams, implementing an ad-hoc version of the decision procedure described here, and later integrating this decision procedure into state-of-the-art SMT solvers. We are also interested in the temporal verification of sequential and concurrent skiplists implementations, including one at the `java.concurrent` standard library. This can be accomplished by the design of verification diagrams that use the decision procedure presented in this paper.

Bibliography

[AC89] Anant Agarwal and Mathews Cherian. Adaptive backoff synchronization techniques. In *Proc. of ISCA 1989*, pages 396–406, 1989.

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[And89] Thomas E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In *Proc. of ICPP 1989*, pages 170–174, 1989.

[ARR03] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.

[AVL62] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[Aya90] Rassul Ayani. Lr-algorithm: concurrent operations on priority queues. In *Proc. of SPDP 1990*, pages 22–25, 1990.

[Bar94] G. Barnes. Wait free algorithms for heaps. Technical report, University of Washington, 1994.

[BB87] Jit Biswas and James C. Browne. Simultaneous update of priority structures. In *Proc. of ICPP 1987*, pages 124–131, 1987.

[BCG01] Guy E. Blelloch, Perry Cheng, and Phillip B. Gibbons. Room synchronizations. In *Proc. of SPAA 2001*, pages 122–133, 2001.

[BCO04] Berdine, Calcagno, and O'Hearn. A decidable fragment of separation logic. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 24, 2004.

[BDES09] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR'09*, pages 178–195, 2009.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BL94] Joan Boyar and Kim S. Larsen. Efficient rebalancing of chromatic search trees. *J. Comput. Syst. Sci.*, 49(3):667–682, 1994.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer-Verlag, 2007.

[BMS95] Anca Browne, Zohar Manna, and Henny B. Sipma. Generalized verification diagrams. In *Proc. of FSTTCS'95*, volume 1206 of *LNCS*, pages 484–498. Springer, 1995.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proc. of VMCAI 2006*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[BNR08] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *Proc. of ECOOP'08*, pages 387–411. Springer, 2008.

[BPZ05] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *Proc. of VMCAI 2005*, pages 164–180, 2005.

[BR06] Jesse D. Bingham and Zvonimir Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Proc. of VMCAI 2006*, pages 207–221, 2006.

[BRS99] Michael Benedikt, Thomas W. Reps, and Shmuel Sagiv. A decidable logic for describing linked data structures. In *Proc. of ESOP 1999*, pages 2–19, 1999.

[BS77] Rudolf Bayer and Mario Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[BSST08] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesar Tinelli. *Handbook of Satifiability*, chapter Satisfiability Modulo Theories. IOS Press, 2008.

[Bur72] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[Cra93] Travis Craig. Building fifo and priority-queuing spin locks from atomic swap. Technical report, University of Washington, Department of Computer Science, 1993.

[DN03] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proc. of VMCAI 2003*, pages 310–324, 2003.

[DS80] Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Inf. Process. Lett.*, 11(1):1–4, 1980.

[Ell87] Carla Schlatter Ellis. Concurrency in linear hashing. *ACM Trans. Database Syst.*, 12(2):195–217, 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[Fra03] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2003.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[GCPV09] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 16–28. ACM, 2009.

[GGK+83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The nyu ultracomputer - designing an mimd shared memory parallel computer. *IEEE Trans. Computers*, 32(2):175–189, 1983.

[Gre02] Michael Greenwald. Two-handed emulation: how to build non-blocking implementation of complex data-structures using dcas. In *Proc. of PODC 2002*, pages 260–269, 2002.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. of FOCS 1978*, pages 8–21, 1978.

[GT90] Gary Graunke and Shreekant S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. of ASPLOS 1989*, pages 64–75, 1989.

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. of ESOP'08*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. of DISC 2001*, pages 300–314, 2001.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[HF03] Timothy L. Harris and Keir Fraser. Language support for lightweight transactions. In *Proc. of OOPSLA 2003*, pages 388–402, 2003.

[HFM88] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proc. of DISC 2002*, pages 265–279, 2002.

[HLM02] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proc. of DISC 2002*, pages 339–353, 2002.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of ICDCS 2003*, pages 522–529, 2003.

[HLMI03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. of PODC 2003*, pages 92–101, 2003.

[HLS95] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable concurrent counting. *ACM Trans. Comput. Syst.*, 13(4):343–364, 1995.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of ISCA 1993*, pages 289–300, 1993.

[HMPS96] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. Concurrent access of priority queues. *Inf. Process. Lett.*, 60(3):151–157, 1996.

[HS02] Danny Hendler and Nir Shavit. Work dealing. In *Proc. of SPAA 2002*, pages 164–172, 2002.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgran-Kaufmann, 2008.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proc. of SPAA 2004*, pages 206–215, 2004.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[HW91] Qin Huang and William E. Weihl. An evaluation of concurrent priority queue algorithms. In *Proc. of SPDP 1991*, pages 518–525, 1991.

[HY86] Meichun Hsu and Wei-Pang Yang. Concurrent operations in extendible hashing. In *Proc. of VLDB 1986*, pages 241–247, 1986.

[IBM] IBM. System/370 principles of operation. order number ga22-7000.

[IBM03] IBM. Powerpc microprocessor family: Programming environments manual for 64 and 32-bit microprocessors, version 2.0, 2003.

[III86] Eugene D. Brooks III. The butterfly barrier. *Int. J. Parallel Program.*, 15(4):295–307, 1986.

[Int94] Intel. Pentium processor family user's manual: Vol 3, architecture and programming manual, 1994.

[IR93] Amos Israeli and Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *Proc. of WDAG 1993*, pages 1–17, 1993.

[JJKS97] Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proc. of PLDI 1997*, pages 226–236, 1997.

[Joh91] T. Johnson. A highly concurrent priority queue based on the b-link tree. Technical report, University of Florida, 1991.

[Jou02] Yuh-Jzer Joung. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Distributed Computing*, 15(3):155–175, 2002.

[Kan89] G. Kane. *MIPS RISC Architecture*. Prentice-Hall, Inc., New York, USA, 1989.

[Kes83] Joep L. W. Kessels. On-the-fly optimization of data structures. *Commun. ACM*, 26(11):895–901, 1983.

[KL80] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[KM99] Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In *Proc. of PODC 1999*, pages 23–32, 1999.

[KMH97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and recovery in generalized search trees. In *Proc. of SIGMOD 1997*, pages 62–72, 1997.

[KNR05] Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *CADE'05*, pages 260–277, 2005.

[Knu68] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 1968.

[KR81] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[KSUH93] Orran Krieger, Michael Stumm, Ronald C. Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *Proc. of ICPP 1993*, pages 201–204, 1993.

[Kum90] Vijay Kumar. Concurrent operations on extendible hashing and its performance. *Commun. ACM*, 33(6):681–694, 1990.

[LAA87] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[Lar01] Kim S. Larsen. Relaxed multi-way trees with group updates. In *Proc. of PODS 2001*, pages 93–101, 2001.

[Lar02] Kim S. Larsen. Relaxed red-black trees with group updates. *Acta Inf.*, 38(8):565–586, 2002.

[Lea99] D. Lea. *Concurrent Programming in JAVA(TM): Design Principles and Pattern*. Addison-Wesley, 1999.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98*, pages 144–153. ACM, 1998.

[LF95] Kim S. Larsen and Rolf Fagerberg. B-trees with relaxed balance. In *Proc. of IPPS 1995*, pages 196–202, 1995.

[Lit80] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of VLDB 1980*, pages 212–223, 1980.

[LQ06] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *Proc. of POPL 2006*, pages 115–126, 2006.

[LQ08] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *Proc. of POPL'08*, pages 171–182. ACM, 2008.

[Man84] Udi Manber. On maintaining dynamic information in a concurrent environment. In *Proc. of STOC 1984*, pages 273–278, 1984.

[McM99] Kenneth L. McMillan. Circular compositional reasoning about liveness. In *Proc. of CHARME'99*, volume 1703 of *LNCS*, pages 342–345. Springer, 1999.

[MCS91a] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[MCS91b] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. of PPOPP 1991*, pages 106–113, 1991.

[Mic02a] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. of SPAA 2002*, pages 73–82, 2002.

[Mic02b] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proc. of PODC 2002*, pages 21–30, 2002.

[ML92] C. Mohan and Frank E. Levine. Aries/im: An efficient and high concurrency index management method using write-ahead logging. In *Proc. of SIGMOD 1992*, pages 371–380, 1992.

[MLH94] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of IPPS 1994*, pages 165–171, 1994.

[MMS02] P. Martin, M. Moir, and G. Steele. Dcas-based concurrent deques supporting bulk allocation. Technical report, Sun Microsystems Laboratories, 2002.

[MN05] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Proc. of CAV 2005*, pages 476–490, 2005.

[Mot86] Motorola. Mc68020 32-bit microprocessor user's manual, 1986.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer, 1995.

[MS95] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical report, University of Rochester, 1995.

[MS98] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[MS07] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications*, pages 47–14 — 47–30, 2007. Chapman and Hall/CRC Press.

[Nel83] Greg Nelson. Verifying reachability invariants of linked structures. In *Proc. of POPL 1983*, pages 38–47, 1983.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NSS91] Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proc. of PODS 1991*, pages 192–198, 1991.

[NSSW87] Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Concurrency control in database structures with relaxed balance. In *Proc. of PODS 1987*, pages 170–176, 1987.

[Opp80] Derek C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, 1980.

[PLJ91] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. A non-blocking algorithm for shared queues using compare-and-swap. In *Proc. of ICPP 1991*, pages 68–75, 1991.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55–74. IEEE CS Press, 2002.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proc. of MICRO 2001*, pages 294–305, 2001.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of ASPLOS 2002*, pages 5–17, 2002.

[RK88] V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Trans. Computers*, 37(12):1657–1665, 1988.

[RRZ05] Silvio Ranise, Christophe Ringeissen, and Calogero G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In *FROCOS'05*, pages 48–64, 2005.

[RZ06a] Silvio Ranise and Calogero G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *Proc. of SEFM 2006*. IEEE CS Press, 2006.

[RZ06b] Silvio Ranise and Calogero G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *Technical report*, 2006.

[Sag86] Yehoshua Sagiv. Concurrent operations on b*-trees with overtaking. *J. Comput. Syst. Sci.*, 33(2):275–296, 1986.

[Sco02] Michael L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proc. of PODC 2002*, pages 31–40, 2002.

[SDK10] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *Proc. of POPL'10*, pages 199–210. ACM, 2010.

[SI94] CORPORATE SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[SI01] Michael L. Scott and William N. Scherer III. Scalable queue-based spin locks with timeout. In *Proc. of PPOPP 2001*, pages 44–52, 2001.

[Sip99] Henny B. Sipma. *Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems*. PhD thesis, Stanford University, 1999.

[Sit92] R. Sites. Alpha architecture reference manual, 1992.

[SMC92] M. L. Scott and J. M. Mellor-Crummey. Fast, contention-free combining tree barriers. Technical report, University of Rochester, Rochester, NY, USA, 1992.

[SS03] Ori Shalev and Nir Shavit. Split-ordered lists: lock-free extensible hash tables. In *Proc. of PODC 2003*, pages 102–111, 2003.

[SS10] Alejandro Sánchez and César Sánchez. Decision procedures for the temporal verification of concurrent lists. In *Proc. of ICFEM'10*, volume 6447 of *LNCS*, pages 74–89. Springer, 2010.

[SS11] Alejandro Sánchez and César Sánchez. A theory of skiplists with applications to the verification of concurrent datatypes. In *Proc. of NFM'11*, volume 6617 of *LNCS*, pages 343–358. Springer, 2011.

[ST97a] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory Comput. Syst.*, 30(6):645–670, 1997.

[ST97b] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[ST03] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proc. of IPDPS 2003*, page 84, 2003.

[SZ96] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 45(4):385–428, 1996.

[SZ99] Nir Shavit and Asaph Zemach. Scalable concurrent priority queue algorithms. In *Proc. of PODC 1999*, pages 113–122, 1999.

[Tar51] Alfred Tarski. A decision method for elementary algebra and geometry. *University of California Press*, 1951.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, 1986.

[TZ04] Cesare Tinelli and Calogero G. Zarba. Combining decision procedures for sorted theories. In *JELIA'04*, volume 3229 of *LNCS*, pages 641–653. Springer, 2004.

[Vaf07] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. of PODC 1995*, pages 214–222, 1995.

[VHHS06] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proc of PPoPP'06*, pages 129–136. ACM, 2006.

[WPK09] Thomas Wies, Ruzica Piskac, and Viktor Kuncak. Combining theories with shared set operations. In *Proc. of Frontiers of Combining Systems (FroCoS'09)*, volume 5749 of *LNCS*, pages 366–382. Springer, 2009.

[YRS+06] Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In *FOSSACS'06*, pages 94–110, 2006.

[YTL87] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Computers*, 36(4):388–395, 1987.

[Zar03] Calogero G. Zarba. Combining sets with elements. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 762–782. Springer, 2003.